Universidad de Valladolid

E.U. de Informática (Segovia)

Grado en Ingeniería Informática de Servicios y Aplicaciones

HCWBB
(Highly Concurrent Web Based Bingo)

Alumno: Manuel Rico López

Tutor: Aníbal Bregón Bregón

Índice:

Índice de imágenes:

Capítulo 1:

Capítulo 2:

Capítulo 3

Capítulo 4

Capítulo 5

Capítulo 6

Capítulo 7

Índice de tablas:

Capítulo 1.

Capítulo 2.

Capítulo 3.

Capítulo 4.

CAPÍTULO 1

INTRODUCCIÓN

Capítulo 1 - Introducción

En este primer capítulo de la documentación del proyecto se explicarán los motivos por los que se decidió desarrollarlo, con qué propósito para cumplir unos objetivos y acabaremos haciendo un resumen de lo que veremos a lo largo de la documentación.

1.1 – Motivación:

Los sistemas distribuidos online están cada vez más solicitados, el número de usuarios que usa este tipo de sistemas aumenta año tras año. En el mundo en que vivimos donde los usuarios tienen un mayor conocimiento tecnológico, este tipo de sistemas distribuidos tiene que dar servicio cada vez a un número mayor de usuarios.

Uno de los tipos de servicios que está más en alza hoy en día es el de los videojuegos, donde millones de usuarios pueden estar jugando al mismo tiempo y la empresa debe proporcionarles un servicio las 24 horas del día durante los 7 días de la semana. Esto se traduce en un alto coste para la empresa que debe tener un equipo hardware costoso equipado con potentes servidores para soportar la carga de miles y miles de usuarios realizando peticiones, como puede ser el ejemplo de un juego online, como el bingo.

Es en este punto en el que se va a concentrar el proyecto, donde, a pesar de la existencia de proyectos similares utilizando otras tecnologías distintas a Erlang, la capacidad de manejar un elevado número de jugadores concurrentes así como la actualización de sus navegadores iniciada desde el lado servidor (sin emplear un mecanismo de 'polling[1]' en el código cliente) harán de la implementación de nuestro bingo online una excelente demostración de cómo combinar paralelismo y software en tiempo semi-real con innovadoras tecnologías web como Websockets[2].

1.2 – Propósito general:

En este proyecto se pretende que miles de jugadores puedan jugar al bingo online usando como servidor un ordenador doméstico que ofrezca un rendimiento de nivel medio. Para ello se diseñará e implementará:

- Una aplicación Erlang capaz de modelar el comportamiento de un bingo utilizando procesos de la máquina virtual de Erlang para modelar los distintos actores del sistema.

- Un sencillo interfaz web para poder utilizar la aplicación usando HTML y JAVASCRIPT.

- Obtener unas medidas de rendimiento del sistema que demuestren su alta capacidad y concurrencia.

[1] **Polling** en computación hace referencia a una operación de consulta constante.

[2] **WebSocket** es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP.

- Diseño de un API HTTP robusto que permita el empleo de la aplicación desde un terminal sin capacidades gráficas en un marco de jugabilidad aceptable.

Para esto, implementaremos una aplicación Erlang OTP modelando la lógica del juego y el estado de la partida. La aplicación hará uso de un servidor web Cowboy con soporte de conexiones empleando un protocolo basado en Websockets para servir al lado cliente el código HTML y un controlador JavaScript para el interfaz así como para mantenerlo actualizado con los cambios más recientes en la partida.

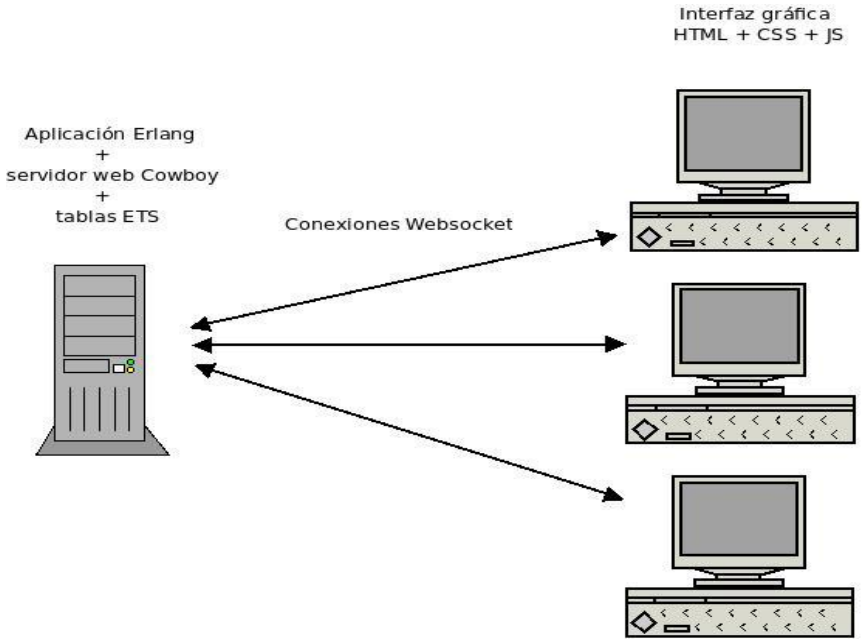


Imagen 1. Arquitectura Inicial

1.3 – Objetivos:

El objetivo principal consistirá en la realización de un juego bingo online en el que usando un ordenador de rendimiento medio como servidor puedan jugar diez mil usuarios de forma concurrente. Más en concreto, este objetivo general se puede dividir en los siguientes subobjetivos:

- El acceso a la aplicación por parte de un importante número de jugadores online al mismo tiempo.

- Que los jugadores puedan registrarse con un nombre de usuario en las partidas en las que va a jugar.

- Que los usuarios tengan un log donde se les informe de todo lo que ocurre a lo largo de las partidas.

- La posibilidad de cantar línea con su respectiva comprobación por parte del servidor.

- La posibilidad de cantar bingo y ganar la partida con su respectiva comprobación por parte del servidor.

1.4 – Entorno Tecnológico:

En cuanto a las tecnologías utilizadas en el proyecto, podemos destacar varias para las diferentes partes de este, separando por tecnologías del lado servidor y tecnologías del lado cliente:

En el lado servidor destacamos:

- Servidor Web Cowboy: responsable de la transmisión de información entre el servidor y los múltiples usuarios concurrentes que forman la partida.

- Aplicación Erlang/OTP: responsable de la implementación de la lógica del juego y del modelado de los distintos navegadores conectados a una partida como actores concurrentes en el sistema.

- Tablas ETS: se explica la ausencia de una base de datos ya que se usan tablas ETS en el código Erlang.

En el lado cliente se usa:

- HTML + CSS: responsables de proveer al usuario de un mínimo interfaz para el seguimiento de la partida y la interacción con la misma.

- JavaScript: responsable del establecimiento de la conexión Websocket con el servidor y de manejar las actualizaciones de estado recibidas del mismo.

1.5 – Organización:

La documentación ha sido estructurada de la siguiente forma:

En el primer capítulo se ha realizado una presentación del proyecto, definiendo los conceptos importantes por los que se caracteriza la aplicación. De este modo queda de una forma clara definida la idea del proyecto con unos objetivos definidos que se irán desarrollando a lo largo de esta documentación.

Seguidamente, en el segundo capítulo denominado estado del arte, mostramos el origen de la idea de la aplicación desarrollada. Además de una explicación en profundidad de las herramientas utilizadas, junto al desarrollo de un nuevo proyecto desde cero.

En tercer lugar, el capítulo plan de proyecto, donde se realiza toda la planificación del proyecto, comenzando por los requisitos principales, seguido por estimaciones de tiempo, recursos, herramientas, costes, etc.

A continuación, el capítulo de análisis, en el cual se realiza el estudio más exhaustivo de los requisitos con sus correspondientes tablas explicativas y todos los diagramas de planificación del proyecto (módulos, casos de uso, secuencia,…).

En quinto lugar, en el capítulo de diseño, se realizara como su título indica, el diseño completo de la aplicación, desde su arquitectura, sus diagramas finales y el diseño de datos e interfaces.

En sexto lugar, realizaremos el capítulo de pruebas en el cual se realizan pruebas de caja negra y blanca y mostraremos las **pruebas de carga** que demuestran el potencial del sistema al ser creado con tecnología servidor Erlang incluso usando un ordenador doméstico..

Nuevamente, en séptimo lugar, desarrollamos el manual para jugar una partida de bingo online, donde se detallará el funcionamiento de dicha aplicación para los usuarios.

Por último, en el octavo capítulo, explicaremos las futuras ampliaciones posibles para la aplicación, además de las conclusiones personales acerca del desarrollo del proyecto.

Finalizando la documentación, hablaremos de la bibliografía utilizada para el desarrollo general tanto de la aplicación como de la documentación.

CAPÍTULO 2

ESTADO DEL ARTE

Capítulo 2 – Estado del Arte

A continuación en este segundo capítulo conoceremos el origen de Erlang y su papel en la actualidad, cómo se instala, qué herramientas se pueden utilizar para desarrollar proyectos con Erlang, cómo funciona la comunicación entre procesos así como la explicación de dos de los más importantes comportamientos (behaviours) y finalizaremos enumerando las buenas prácticas a la hora de desarrollar software con Erlang.

2.1 - Historia de Erlang:

La idea de Erlang surgió por la necesidad de Ericsson de acotar un problema que había surgido en su plataforma AXE, que estaba siendo desarrollada en PLEX, un lenguaje propietario. Joe Amstrong junto a Elshiewy y Robert Virding desarrollaron una lógica concurrente de programación para canales de comunicación. Esta álgebra de telefonía permitía a través de su notación describir el sistema público de telefonía (POTS) en tan sólo quince reglas.

A través del interés de llevar esta teoría a la práctica desarrollaron modelos en Ada, CLU, Smalltalk y Prolog entre otros. Así descubrieron que el álgebra telefónica se procesaba de forma muy rápida en sistemas de alto nivel, es decir, en Prolog, con lo que comenzaron a desarrollar un sistema determinista en él.

La conclusión a la que llegó el equipo fue que, si se puede resolver un problema a través de una serie de ecuaciones matemáticas y portar ese mismo esquema a un programa de forma que el esquema funcional se respete y entienda tal y como se formuló fuera del entorno computacional, puede ser fácil de tratar por la gente que entiende el esquema, incluso mejorarlo y adaptarlo.

Prolog no era un lenguaje pensado para concurrencia, por lo que se decidieron a realizar uno que satisfaciera todos sus requisitos, basándose en las ventajas que habían visto de Prolog para conformar su base. Erlang vió la luz en 1986, después de que Joe Amstrong desarrollase la idea base como intérprete sobre Prolog, con un número reducido de instrucciones que rápidamente fue creciendo gracias a su buena acogida. Básicamente, los requisitos que se buscaban cumplir eran:

- Los procesos debían de ser una parte intrínseca del lenguaje, no una librería o framework de desarrollo.

- Debía poder ejecutar desde miles a millones de procesos concurrentes y cada proceso ser independiente del resto, de modo que si alguno de ellos se corrompiese no dañase el espacio de memoria de otro proceso. Este requisito nos lleva a que el fallo de los procesos debe de ser aislado del resto del programa.

- Debe poder ejecutarse de modo ininterrumpido, lo que obliga a que para actualizar el código del sistema no se deba detener su ejecución, sino que se recargue en caliente.

En 1989, el sistema comenzaba a dar sus frutos pero con un rendimiento insuficiente, necesitaban que fuese al menos unas cuarenta veces más rápido. Es entonces cuando Mike Williams escribió el emulador, cargador, planificador y recolector de basura (en lenguaje C) mientras que Joe Amstrong escribía el compilador, la estructura de datos, el heap de memoria y la pila; por su parte Robert Virding se encargaba de escribir las librerías. El sistema

desarrollado se optimizó a un nivel en el que consiguieron aumentar su rendimiento 120 veces de lo que hacía el intérprete en Prolog.

En los años 90, tras haber conseguido desarrollar productos de la gama AXE con este lenguaje, se le potenció agregando elementos como distribución, estructura OTP, HiPE (High Performance Erlang) , sintaxis de bit o compilación de patrones para "matching". Erlang comenzaba a ser una gran pieza de software, pero tenía varios problemas para que pudiera ser adoptado de forma amplia por la comunidad de programadores. Desafortunadamente para el desarrollo de Erlang, aquel periodo fue también la década de Java y Ericsson decidió centrarse en lenguajes usados globalmente por lo que prohibió seguir desarrollando Erlang.

Esto provocó que la comunidad de programadores Erlang comenzase a crecer fuera de Ericsson. El equipo OTP se mantuvo desarrollando y soportando Erlang.

Antes de 2010 Erlang agregó capacidad para SMP (Symmetric Multi-Processing) y más recientemente para multi-core. La revisión de 2010 del emulador BEAM se ejecuta con un rendimiento 300 veces superior al de la versión del emulador en C, por lo que es 36.000 veces más rápido que el original interpretado en Prolog. Cada vez más sectores se hacen eco de las capacidades de Erlang y cada vez más empresas han comenzado desarrollos en esta plataforma por lo que se augura que el uso de este lenguaje siga en alza.

2.2 - Introducción a Erlang:

Erlang comienza a ser un entorno y un lenguaje de moda debido a la existencia creciente de empresas orientadas a la prestación de servicios por Internet con un elevado volumen de transacciones (como videojuegos en red o sistemas de mensajería móvil y chat). Existe una necesidad imperiosa de desarrollar entornos con las características de la máquina de Erlang y la metodología de desarrollo proporcionada por OTP.

Erlang fue propietario hasta 1998 momento en que fue cedido como código abierto a la comunidad.

Para comprender qué es Erlang, debemos entender que se trata de un entorno o plataforma de desarrollo completa. Erlang proporciona no sólo el compilador para poder ejecutar el código, sino que posee también una colección de herramientas, y una máquina virtual sobre la que ejecutarlo, por lo tanto existen dos enfoques:

2.2.1 – Erlang como lenguaje:

Erlang podría catalogarse como un lenguaje híbrido, ya que contiene elementos de tipo funcional, de tiempo imperativo, e incluso algunos rasgos que permiten cierta orientación a objetos, aunque no completamente.

Donde podríamos encajar mejor a Erlang es como lenguaje orientado a la concurrencia. Tiene una gran facilidad para la programación distribuida, paralela o concurrente y además con mecanismos para la tolerancia de fallos.

Desde un inicio fue diseñado para ejecutarse de forma ininterrumpida, lo que significa que se puede cambiar el código de las aplicaciones sin detener su ejecución.

2.2.2 – Erlang como entorno de ejecución:

Como se mencionó antes, Erlang es una plataforma de desarrollo que proporciona no sólo un compilador, sino también una máquina virtual para su ejecución. A diferencia de otros lenguajes interpretados como Python, Perl, PHP o Ruby, Erlang se pseudocompila y su máquina virtual le proporciona una importante capa de abstracción que le dota de la capacidad de manejar y distribuir procesos entre nodos de forma totalmente transparente, sin necesidad de usar librerías específicas.

La máquina virtual sobre la que se ejecuta el código pseudocompilado de Erlang, que le proporciona todas la características de distribución y comunicación de procesos, es también una máquina que interpreta un pseudocódigo máquina que nada tiene que ver, a ese nivel, con el lenguaje Erlang. Esto ha permitido la proliferación de los lenguajes que emplean la máquina virtual pero no el lenguaje en sí, como pueden ser: Reia, Elixir, Efene, Joxa, etc...

2.3 – Características:

Los desarrolladores de Erlang buscaron lo que sería un sistema de desarrollo óptimo basado en las siguientes premisas:

- **Distribuido:**

El sistema debía de ser distribuido para poder balancear su carga entre los sistemas hardware. Se buscaba un sistema que pudiera lanzar procesos no sólo en la máquina en la que se ejecuta, sino que también fuera capaz de hacerlo en otras máquinas. Lo que en lenguajes como C viene a ser PVM o MPICH pero sin el uso explícito de ninguna librería.

- **Tolerante a fallos:**

Si una parte del sistema tiene fallos y tiene que detenerse, que esto no signifique que todo el sistema se detenga.

En sistemas software como PLEX o C, un fallo en el código determina una interrupción completa del programa con todos sus hilos y procesos. Hay otros lenguajes como Java, Python o Ruby que manejan estos errores como excepciones, afectando sólo a una parte del programa y no a todos sus hilos. No obstante, en los entornos con memoria compartida, un error puede dejar corrupta esta memoria por lo que esa opción no garantiza tampoco que no afecte al resto del programa.

- **Escalable:**

Los sistemas operativos convencionales tenían problemas en mantener un elevado número de procesos en ejecución. Los sistemas de telefonía que desarrolla Ericsson se basan en tener un proceso por cada llamada entrante, que vaya controlando los estados de la misma y pueda provocar eventos hacia un manejador, a su vez con sus propios procesos. Por lo que se buscaba un sistema que pudiese gestionar desde cientos de miles, hasta millones de procesos.

- **Cambiar el código en caliente:**

También es importante en el entorno de Ericsson, y en la mayoría de sistemas críticos o sistemas en producción de cualquier índole, que el sistema no se detenga nunca, aunque haya que realizar actualizaciones. Por ello se agregó también como característica el hecho de que el código pudiese cambiar en caliente, sin necesidad de parar el sistema y sin que afectase al código en ejecución.

- **Asignaciones únicas:**

Como en los enunciados matemáticos la asignación de un valor a una variable se hace una única vez y, durante el resto del enunciado, esta variable mantiene su valor inmutable. Esto nos garantiza un mejor seguimiento del código y una mejor detección de errores.

- **Lenguaje simple:**

Para rebajar la curva de aprendizaje el lenguaje debe de tener pocos elementos y ninguna excepción. Erlang es un lenguaje simple de comprender y aprender, ya que tiene nada más que dos estructuras de control, carece de bucles y emplea técnicas como la recursividad y modularización para conseguir algoritmos pequeños y eficientes. Las estructuras de datos se simplifican también bastante y su potencia, al igual que en lenguajes como Prolog o Lisp, se basa en las listas.

- **Orientado a la concurrencia:**

Como una especie de nueva forma de programar, este lenguaje se orienta a la concurrencia de manera que las rutinas más íntimas del propio lenguaje están preparadas para facilitar la realización de programas concurrentes y distribuidos.

- **Paso de mensajes en lugar de memoria compartida:**

Uno de los problemas de la programación concurrente es la ejecución de secciones críticas de código para acceso a porciones de memoria compartida. Este control de acceso acaba siendo un cuello de botella ineludible. Para simplificar e intentar eliminar el máximo posible de errores, Erlang/OTP se basa en el paso de mensajes en lugar de emplear técnicas como semáforos o monitores. El paso de mensajes hace que un proceso sea el responsable de los datos y la sección crítica se encuentre sólo en este proceso, de modo que cualquiera que pida ejecutar algo de esa sección crítica, tenga que solicitárselo al proceso en cuestión. Esto abstrae al máximo la tarea de desarrollar programas concurrentes, simplificando enormemente los esquemas y eliminando la necesidad del bloque explícito.

En sí, los lenguajes más difundidos hoy en día, como C# o Java, presentan el problema de carecer de elementos a bajo nivel integrados en sus sistemas que les permitan desarrollar aplicaciones concurrentes de forma fácil. Esta es la razón de que en el mundo Java comience a hacerse cada vez más visible un lenguaje como Scala.

2.4 – Erlang en el mundo:

Los desarrollos en Erlang cada vez son más visibles para todos en cuanto se trata de concurrencia y gestión masiva de eventos o elementos sin saturarse ni caer. Esto es un punto esencial y decisivo para empresas que tienen su nicho de negocio en Internet y que han pasado de vender productos a proveer servicios a través de la red.

En esta sección veremos la influencia de Erlang y cómo se va asentando en el entorno empresarial y en las comunidades de software libre y el tipo de implementaciones que se realizan en uno y otro ámbito.

2.4.1 – Sector empresarial:

Debido a las ventajas intrínsecas del lenguaje y su entorno se ha hecho patente la creación de modelos reales MVC (Model View Controller) para desarrollo web. Merecen mención elementos tan necesarios como ChicagoBoss[3] o Nitrogen[4], cuyo uso pueden verse en empresas como la española Tractis.

También es conocido el caso de Facebook que emplea Erlang en su implementación de chat para soportar los mensajes de sus 70 millones de usuarios. Al igual que Tuenti que también emplea esta tecnología.

La empresa inglesa Demonware, especializada en el desarrollo y mantenimiento de infraestructura y aplicaciones servidoras para videojuegos en Internet, comenzó a emplear Erlang para poder soportar el número de jugadores de títulos tan afamados como Call of Duty.

Varias empresas del sector del entretenimiento que fabrican aplicaciones móviles también se han sumado a desarrollar sus aplicaciones de parte servidora en Erlang/OTP.

Un ejemplo es WhatsApp, la aplicación actualmente más relevante para el intercambio y envío de mensajes entre smartphones emplea a nivel de servidor sistemas desarrollados en Erlang.

Desde que surgió el modelo Cloud, cada vez más empresas de software están prestando servicios online en lugar de vender productos, por lo que se enfrentan a un uso masificado por parte de sus usuarios, e incluso a ataques de denegación de servicio. Estos escenarios junto con servicios bastante pesados e infraestructuras no muy potentes hacen cada vez más necesarias herramientas como Erlang.

Una de las mejores pruebas de que Erlang/OTP funciona, es mostrar las comparaciones que empresas como Demonware o gente como el propio Joe Armstrong han realizado. Sistemas sometidos a un banco de pruebas para comprobar cómo rinden en producción real o cómo podrían rendir en entornos de pruebas controlados.

En el caso previamente nombrado de Demonware, podríamos decir que en 2005 construyeron su infraestructura en C++ y MySQL cuando su concurrencia de usuarios no superaba los 80 jugadores y su código se colgaba con frecuencia, lo que suponía un grave

[3] Web Framework de Erlang (http://www.chicagoboss.org/).

[4] Web Framework de Erlang (http://nitrogenproject.com/).

problema. En 2006 se reescribió toda la lógica de negocio en Python. Manteniendo a nivel interno C++ con lo que el código se había hecho difícil de mantener. Finalmente en 2007, se reescribió el código de los servidores de C++ con Erlang. Fueron unos 4 meses de desarrollo con el que se consiguió que el sistema ya no se colgase, que se mejorase y facilitase la configuración del sistema (en C++ era necesario reiniciar para reconfigurar, lo que implicaba desconectar a todos los jugadores). Para entonces ya se habían llegado a los veinte mil usuarios concurrentes.

A finales de 2007 llegó Call of Duty 4, que supuso un crecimiento constante de usuarios durante 5 meses continuados. Se pasó de 20 mil a 2,5 millones de usuarios. De 500 a 50 mil peticiones por segundo. La empresa tuvo que ampliar su nodo de 50 a 1850 servidores en varios centros de datos. Lo que sin Erlang no habría sido posible de afrontar para la empresa ya que la inversión en un número mayor de servidores los hundiría económicamente, no sería viable.

Demonware es una de las empresas que ha visto las ventajas de Erlang. La forma en la que implementa la programación concurrente y la gran capacidad de escalabilidad. Gracias a estos factores, han podido estar a la altura de prestar el servicio de los juegos en línea más usados y jugados de los últimos tiempos.

Una vez explicada la experiencia de Demonware, podemos observar un banco de pruebas realizado por Joe Amstrong sobre un servicio empleando un par de configuraciones de Apache y Yaws.

La prueba es bastante fácil, de un lado, un servidor, de otro, un cliente para medición y 14 clientes para generar carga.

Esta consistía en generar un ataque de denegación de servicio (DoS), que hiciera que los servidores web, al recibir un número de peticiones excesivo, fuesen degradando su servicio hasta dejar de darlo. Es bien conocido que este hecho pasa con todos los sistemas, ya que los recursos de un servidor son finitos. No obstante, por su programación, pueden pasar cosas como las que se visualizan en el gráfico:

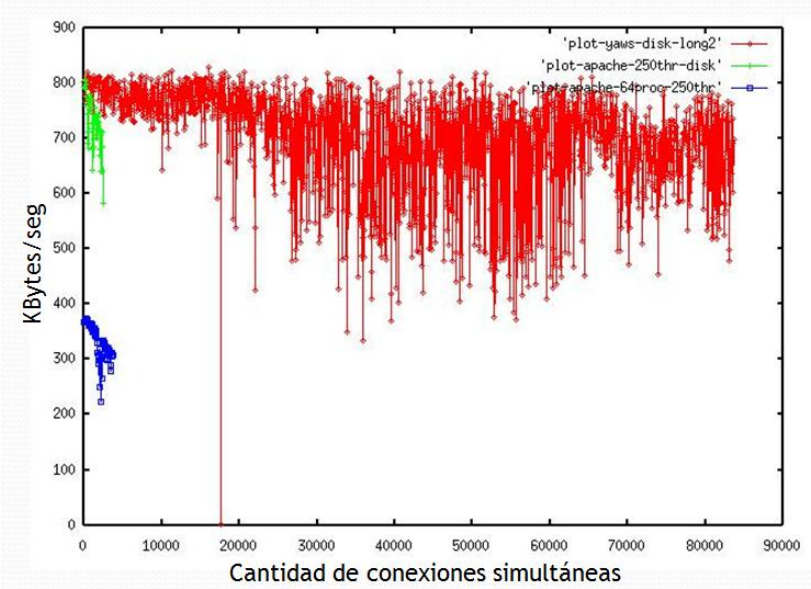


Imagen 2. Yaws vs Apache

En este caso, pasa algo parecido a lo visto con Demonware anteriormente durante su expansión, Apache no puede procesar más de 4000 peticiones simultáneas, en parte debido a su integración íntimamente ligada al sistema operativo, que le limita. Sin embargo, Yaws se mantiene con el mismo rendimiento hasta llegar a superar las 80 mil peticiones simultáneas.

Erlang está construido con gestión de procesos propia y desligada del sistema operativo. En sí, suele ser más lenta que la que proporciona el sistema operativo, pero sin duda la escalabilidad y el rendimiento que se consigue pueden paliar ese hecho. Cada nodo de Erlang puede manejar en total unos 2 millones de procesos.

2.4.2 – Software Libre:

Hay muchas muestras de proyectos de gran envergadura de muy diversa índole creados en base a Erlang. La mayoría de ellos se centra en entornos en los que se saca gran ventaja de la gestión de concurrencia y distribución que realiza el sistema de Erlang.
Nombraremos algunos ejemplos conocidos:

- **Bases de Datos Distribuidas:**
  - Apache CouchDB: es una base de datos documental con acceso a datos mediante HTTP y empleando el formato REST. Es uno de los proyectos que están acogidos en la fundación Apache.
  - Riak: una base de datos NoSQL inspirada en Dynamo (la base de datos NoSQL de Amazon). Es usada por empresas como Mozilla y Comcast. Se basa en una distribución de fácil escalado y completamente tolerante a fallos.
- **Servidores Web:**
  - Yaws: como servidor web completo, con posibilidad de instalarse y configurarse para ello. Su configuración se realiza de forma bastante similar a Apache. Tiene unos scripts que se ejecutan a nivel de servidor bastante potentes y permite el uso de CGI y FastCGI.
  - Cowboy: servidor pequeño, rápido y modular para desarrollar con Erlang. Usa behaviours para extender funcionalidad. La librería de cowboy se define como un pool de servidores TCP. Por defecto aporta controladores HTTP para: request, websockets, static y REST; aunque también se pueden crear otros controladores para otros servidores TCP como SMTP.
- **Frameworks Web:**
  - ChicagoBoss : quizás el más activo y completo de los frameworks web para Erlang a día de hoy. Tiene implementación de vistas, plantillas (ErlyDTL), definición de rutas, controladores y modelos a través de un sistema ORM.

- Erlang Web: es un sistema desarrollado por Erlang Solutions que trata igualmente las vistas y la parte del controlador pero no la parte de la base de datos.

- **CMS(** Content Management System):

  - Zotonic: sistema CMS que permite el diseño de páginas web de forma sencilla a través de la programación de las vistas (DTL) y la gestión del contenido multimedia, texto y otros aspectos a través del interfaz de administración.

- **Chat:**

  - ejabberd: servidor de XMPP muy utilizado en el mundo Jabber. Este servidor permite el escalado y la gestión de multi-dominios. Es usado en sitios como la BBC Radio LiveText, Ovi de Nokia, KDE Talk, Chat de Facebook, Chat de Tuenti, LiveJournal Talk, etc.

- **Colas de Mensajes:**

  - RabbitMQ: servidor de cola de mensajes muy utilizado en sistemas de entornos web con necesidad de este tipo de sistemas para conexiones de tipo websocket, AJAX o similar en la que se haga necesario un comportamiento asíncrono sobre las conexiones síncronas. Fue adquirido por SpringSource, una filial de VMWare en abril de 2010.

2.5 – Instalación Erlang:

Tener la máquina virtual de Erlang operativa con todas sus características es bastante fácil gracias a la gran cantidad de instaladores y distribuciones preparadas que existen en la web. Las versiones oficiales se ofrecen desde la página web oficial de Erlang (http://erlang.org/).

En este apartado veremos como descargar e instalar Erlang, tanto si lo queremos hacer desde paquetes listos para funcionar directamente o desde código fuente.

2.5.1 – Instalación en Windows:

La descarga para Windows se puede realizar desde la web de descargas (http://www.erlang.org/download.html) de la página oficial de Erlang. Entre los paquetes que hay para descargar se puede encontrar Windows Binary File . Se trata de un instalador que nos guiará paso a paso en la instalación.

La instalación en estos sistemas se divide en varios pasos. Se seleccionan los paquetes a instalar y la ruta:

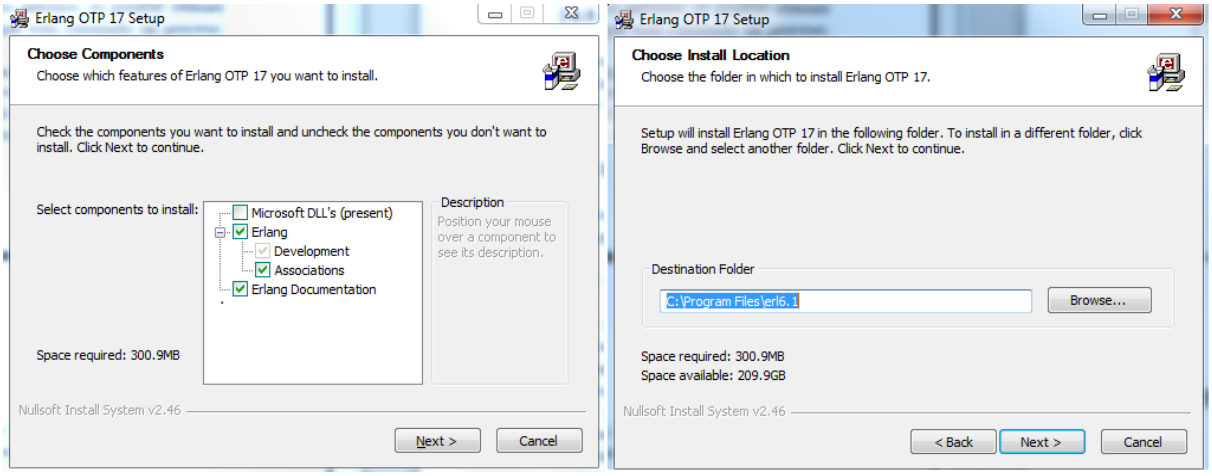


Imagen 3. Instalación Erlang. Primeros pasos

Seguimos con la instalación hasta que el instalador nos informe de que ha finalizado con éxito. En el menú de inicio veremos que se ha creado un nuevo grupo de programas. Podemos lanzar el que tiene como título Erlang con el logotipo del lenguaje a su lado para que se muestre la siguiente pantalla:

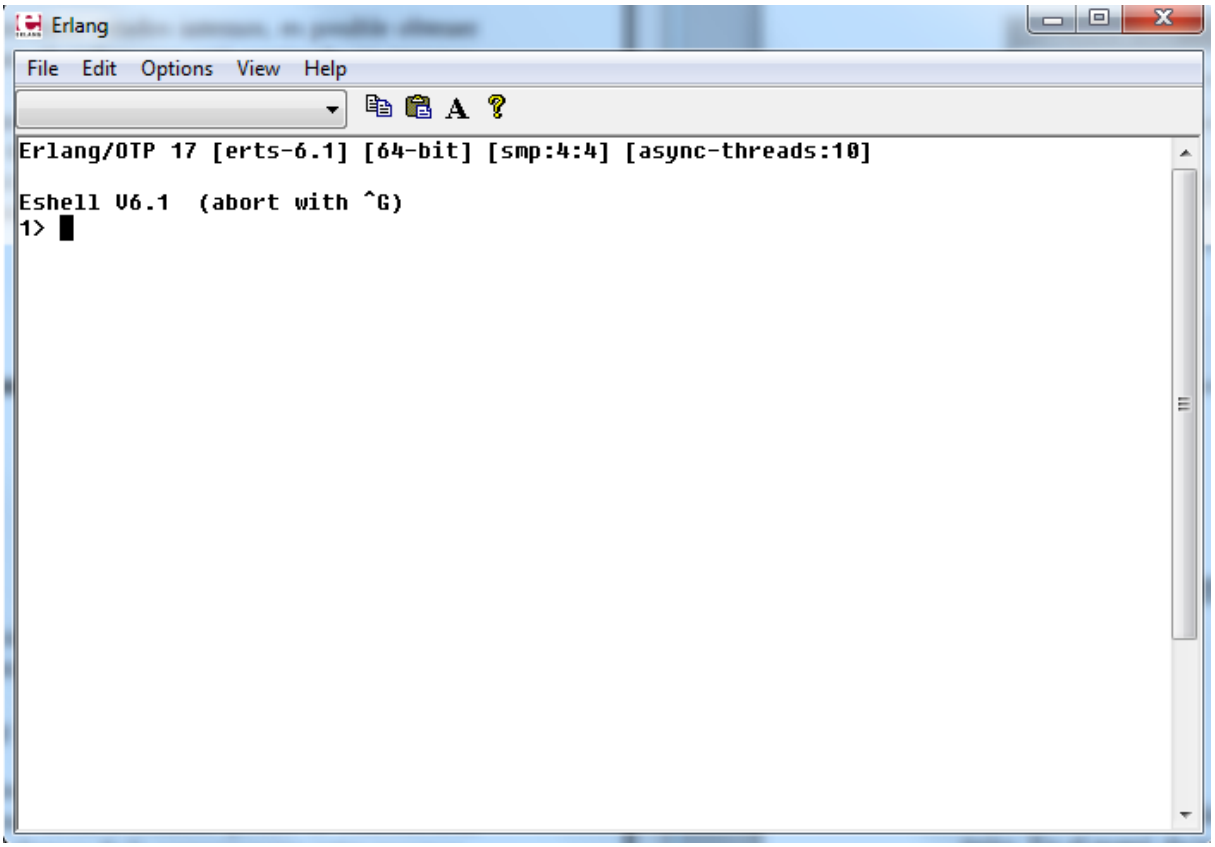


Imagen 4. Consola Erlang

Ahora ya tenemos lista la consola de Erlang. Podemos tomar cualquier ejemplo del libro para probar su funcionamiento.

2.5.2 - Instalación en sistemas GNU/Linux:

La mayoría de distribuciones GNU/Linux disponen de sistemas de instalación de paquetes de forma automatizada. Erlang está disponible por defecto en la mayoría de estas distribuciones pero, dado que estos paquetes en muchas de estas distribuciones se marcaron como estables hace mucho tiempo las versiones de Erlang disponibles pueden ser algo antiguas.

Por esta razón podemos realizar desde paquetes binarios o compilando el código fuente.

2.5.2.1 – Desde paquetes Binarios:

Tenemos la opción de descargar un paquete actualizado e instalarlo en lugar del que provee por defecto la distribución que estemos usando. Los paquetes actuales se pueden descargar desde la web de Erlang Solutions:

https://www.erlang-solutions.com/downloads/download-erlang-otp

Las versiones para CentOS y Fedora se descargan en formato RPM y pueden instalarse a través de la herramienta rpm.

Las versiones para Debian, Ubuntu y Raspbian se descargan en formato DEB y pueden instalarse a través de la herramienta dpkg.

Una vez instalado podemos ejecutar desde consola el comando *erl* o *erlc* entre otros.

2.5.2.2 – Compilando el Código Fuente:

Otra opción es compilar el código fuente para los sistemas en los que no se encuentre Erlang en la última versión o se quiera disponer de varias versiones instaladas en rutas diferentes a las que se establecen por defecto.

En este caso habrá que descargar el último archivo comprimido de código:

```
# wget http://www.erlang.org/download/otp_src_R15B02.tar.gz
# tar xzf otp_src_R15B02.tar.gz
# cd otp_src_R15B02
# ./configure
# make && make install
```

Imagen 5. Compilar desde GNU/Linux

La compilación requiere que se disponga en el sistema de un compilador y las librerías en las que se basa Erlang. El comando configure nos dará pistas sobre las librerías que haya que instalar. Hay que recordar que sistemas como Ubuntu no disponen de acceso directo como usuario root. En su lugar se debe de acceder a root a través del comando sudo.

2.5.3 – Otros sistemas:

La empresa Erlang Solutions provee paquetes de instalación para otros sistemas como MacOS X. En este sistema podemos optar por instalar este paquete o por la instalación desde otros sistemas como MacPorts (http://www.macports.org/).

En sistemas como OpenSolaris o BSD (FreeBSD, OpenBSD o NetBSD) la solución más común es instalar desde código fuente tal y como se comentó en el punto 2.5.2.2, "Compilando el Código Fuente" .

2.6 – La línea de comandos:

Erlang como máquina virtual dispone de una línea de comandos para facilitar su gestión y demostrar su versatilidad permitiendo conectar una consola a un nodo que se encuentre en ejecución y permitir al administrador obtener información del servidor en ejecución.

La línea de comandos es por tanto uno de los principales elementos de Erlang. En este punto veremos las opciones que nos ofrece este intérprete de comandos para facilitar la tarea de gestión.

2.6.1 – Registros:

En la consola se pueden gestionar los registros a través de las siguientes funciones:

- rd(R,D) : define un registro en la línea de comandos:

```
> rd(state, {hits, miss, error}).
```

- rl() / rl(R) : muestra todos los registros definidos en la línea de comandos en el primer caso y solo el registro pasado como parámetro en el segundo caso. La definición se muestra como se escribiría dentro de un fichero de código.

- rf() / rf(R) : elimina la definición de los registros cargados. La primera forma elimina todos los registros mientras que la segunda solo elimina el registro pasado como parámetro R.

- rr(Modulo) / rr(Wildcard) / rr(MoW,R) / rr(MoW,R,O) : carga los módulos de uno o varios ficheros. Los ficheros se pueden indicar mediante el nombre de un módulo, o el nombre de un fichero o varios con el uso de comodines (wildcard). Se puede agregar un segundo parámetro que indique el registro que se desea cargar y un tercer parámetro que se usará como conjunto de opciones .

2.6.2 – Módulos:

Indicaremos todos los comandos referentes a la compilación, carga e información para los módulos:

- c(FoM) : compila un fichero pasando su nombre como parámetro. El nombre proporcionado será un átomo con el nombre del módulo o una cadena que indique el nombre del fichero, opcionalmente con su ruta.

- l(M) : permite cargar un módulo.

- m() / m(M) : muestra todos los módulos cargados en memoria en el primer caso e información detallada del módulo cargado en el segundo caso. Se muestra información como la fecha y hora de compilación, la ruta de dónde se encuentra el módulo en el sistema de ficheros, las funciones que exporta y las opciones de compilación.

- lc([F]) : lista de ficheros a compilar.

- nl(M) : carga el módulo indicado en todos los nodos conectados.

- nc(FoM) : compila y carga el módulo o fichero en todos los nodos conectados.

- y(F) : genera un analizador Yecc, el fichero pasado como parámetro debe de ser un fichero con sintaxis válida para Yecc.

2.6.3 – Variables:

En la línea de comandos se pueden emplear variables. Estas variables tienen el comportamiento de única asignación igual que el código que podemos escribir en cualquier módulo. Las siguientes funciones nos permiten gestionar estas variables:

- b() : muestra todas las variables empleadas o enlazadas (binding) a un valor en la línea de comandos.

- f() / f(X) : indica a la línea de comandos que olvide (forget) todas las variables o solo la indicada como parámetro.

2.6.4 – Histórico:

La consola dispone de un histórico que nos permite repetir comandos ya utilizados en la consola. El histórico es configurable y contendrá los últimos comandos tecleados. El símbolo del sistema (o prompt) nos indicará el número de orden que estamos ejecutando.

Además de los comandos, la consola de Erlang también almacena los últimos resultados. El número de resultados almacenados también es configurable.

Estas son las funciones que pueden emplearse:

- e(N) : repite el comando con orden N según el símbolo de sistema de la consola.

- h() : muestra el histórico de comandos ejecutados.

- history(N) : configura el número de entradas que serán almacenadas como histórico.

- results(N) : configura el número de resultados que serán almacenados como histórico.

- v(N) : obtiene el resultado de la línea correspondiente pasada como parámetro. A diferencia de e(N) el comando no se vuelve a ejecutar, solo se muestra el resultado del comando N ejecutado anteriormente.

2.6.5 – Procesos:

Estas son funciones rápidas y de gestión sobre procesos:

- bt(Pid) : obtiene el trazado de pila del proceso en ejecución.

- flush() : muestra todos los mensajes enviados al proceso de la consola.

- i(X,Y,Z) : muestra información de un proceso dando sus números como argumentos separados de la función. La información obtenida es el estado de ejecución del proceso, procesos a los que está enlazado, la cola de mensajes, el diccionario del proceso y memoria utilizada entre otras opciones más.

- pid(X,Y,Z) : obtiene el tipo de dato PID de los números dados.

- regs() / nregs() : lista todos los procesos registrados (con nombre) en el nodo actual o en todos los nodos conectados respectivamente.

- catch_exception(B) : cada ejecución se realiza mediante un evaluador. Cuando se lanza una excepción el evaluador es regenerado por el proceso de la consola. Esto provoca que se pierdan tablas ETS entre otras cosas. Si ejecutamos esta función con true el evaluador captura la excepción y no muere.

- i() / ni() : muestra todos los procesos del nodo o de todos los nodos conectados respectivamente.

2.6.6 – Directorio de trabajo:

En cualquier momento podemos modificar el directorio de trabajo dentro de la consola. Las siguientes funciones nos ayudan en esta y otras tareas relacionadas:

- cd(Dir) : cambia el directorio de trabajo. Se indica una lista de caracteres con la ruta relativa o absoluta para el cambio.

- ls() / ls(Dir) : lista el directorio actual u otro indicado como parámetro de forma relativa o absoluta a través de una lista de caracteres.

- pwd() : imprime el directorio de trabajo actual.

2.6.7 – Modo JCL:

Cuando se presiona la combinación de teclas Control+G se accede a una nueva consola. Esta consola es denominada JCL (Job Control Mode o modo de control de trabajos). Este modo nos permite lanzar una nueva consola, conectarnos a una consola remota, detener una consola en ejecución o cambiar de una a otra consola.

Estos son los comandos que podemos emplear en este modo:

- c [nn] : conectar a una consola. Si no se especifica un número vuelve al actual.
- i [nn] : detiene la consola actual o la que corresponda al número que se indique como argumento. Es útil cuando se quiere interrumpir un bucle infinito sin perder las variables empleadas.
- k [nn] : mata la consola actual o la que corresponda al número que se indique como argumento.
- j : lista las consolas en ejecución. La consola actual se indicará con un asterisco (*).
- s [shell] : inicia una consola nueva. Si se indica el nombre de un módulo como argumento se intentará lanzar un proceso con ese módulo como consola alternativa.
- r [node [shell]] : indica que deseamos crear una consola en un nodo al que se tiene conexión. Se lanza una consola en ese nodo y queda visible en el listado de consolas. Se puede indicar también una consola alternativa en caso de disponer de ella.
- q : finaliza la ejecución del nodo Erlang en el que estemos ejecutando el modo JCL.

2.6.8 – Salir de la consola:

Para salir de la consola hay varias formas. Se puede salir presionando dos veces la combinación de teclas Control+C, ejecutando la función de consola q() o a través del modo JCL y su comando q.

2.7 – Desarrollando con Erlang/OTP:

Antes de desarrollar ninguna aplicación distribuida usando Erlang/OTP hay que comprender bien qué es un proceso.

2.7.1 – Procesos:

En Erlang, la unidad de concurrencia es el proceso. Un proceso representa una actividad que se va a llevar a cabo, un agente que corre un bloque de código de un programa, concurrente a otros procesos que corren su propio código, a su propio paso. Los procesos son un poco como la gente, individuos que no comparten nada entre ellos. Con esto no quiero decir que la gente no sea generosa, pero si tú comes una comida, nadie más se llenará comiendo esa comida y, más importante aún, si comes comida en mal estado, solamente tu sentirás malestar. Tú mantienes tus pensamientos en tu mente y los mantienes ahí independientemente de lo que los demás piensen. Así es como se comporta un proceso.

Los procesos están separados de los demás y se garantizan el no molestarse entre ellos aunque su estado interno cambie. Un proceso tiene su propia memoria de trabajo y su propio buzón para los mensajes que le llegarán.

Mientras los hilos en otros lenguajes de programación y sistemas operativos son actividades concurrentes que comparten el mismo espacio de memoria, los procesos de Erlang puede trabajar de forma segura asumiendo que nadie más hurgará en su trabajo ni cambiara sus datos desde el microsegundo uno a los siguientes. Decimos que los procesos encapsulan su estado.

Ya que los procesos no pueden cambiar directamente cada uno de los otros estados internos, es posible obtener una tolerancia a errores alta puesto que no importa como de malo sea el código que está corriendo un proceso, porque no podrá corromper el estado interno de los otros procesos.

Los procesos no comparten datos internos de cada uno, ellos se comunican copiándolo. Es decir, si un proceso quiere intercambiar información con otro proceso, el primero manda un mensaje, este mensaje es una copia de su información interna que el destinatario sólo puede leer. Esta semántica fundamental de paso de mensajes hace de la distribución una parte natural de Erlang. En la vida real tu no puedes compartir información a través de un cable, tu solo puedes copiarla y enviarla. El proceso de comunicación en Erlang siempre funciona como si el destinatario del mensaje recibiera una copia personal del mensaje a través del emisor, incluso si esto ocurre en la misma computadora.

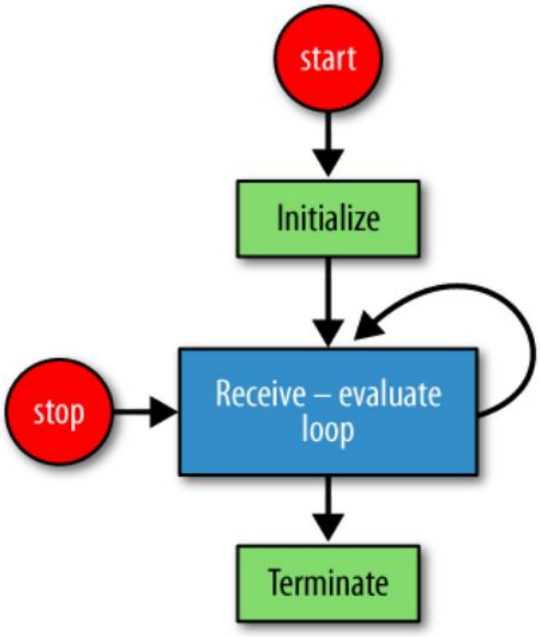


Imagen 6. Ciclo de vida de un proceso Erlang

Esta distribución transparente permite a los programadores Erlang a mirar a la red como una colección de recursos.

El problema principal en todos los sistemas concurrentes, el cual todos los desarrolladores tienen que resolver, es compartir información. Si tu separas un problema grande en distintas

tareas a realizar, ¿cómo se comunicarán estas tareas con las otras? Para resolver este problema hay cuatro paradigmas de la comunicación entre procesos.

- **Shared Memory with Locks.**
- **Software Transactional Memory (STM).**
- **Futures, Promises and Similar.**
- **Message Passing.**

Como dijimos antes, los procesos Erlang se comunican mediante el paso de mensajes, por lo que profundizaremos en el cuarto paradigma.

2.7.2 – Message Passing (Paso de Mensajes).

El paso de mensajes significa que la información que reciben los procesos de forma correcta es una copia de la información original que contiene el proceso emisor y que nada de lo que se haga con la copia de información es observable para este. La única forma de devolver la información al proceso emisor, es enviar otro mensaje en la dirección inversa. Como dijimos en el apartado 2.7.1 “Procesos”, una de las consecuencias más importantes es que la comunicación entre procesos funciona igual si el emisor y receptor están en la misma computadora o separados en red.

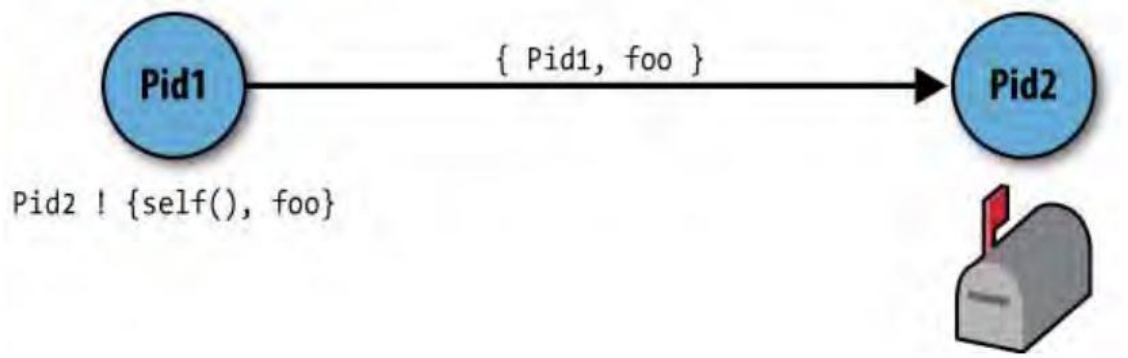


Imagen 7. Paso de mensajes entre procesos

El paso de mensajes puede efectuarse de dos maneras:

- **Paso de mensajes síncrono:** el emisor no puede hacer nada hasta que el mensaje no haya sido recibido por el receptor final (en el mundo real, la comunicación síncrona entre máquinas separadas solo es posible si el receptor envía un mensaje de confirmación al emisor, diciendo que todo se ha producido correctamente y que el emisor puede continuar con su ejecución, pero este detalle puede mantenerse oculto al programador).

- **Paso de mensajes asíncrono:** el emisor puede proceder inmediatamente después de enviar el mensaje.

En Erlang, las primitivas de paso de mensajes son asíncronas, porque es más fácil de implementar que la forma síncrona, de este modo se evita hacer que el receptor siempre envíe una respuesta explícita de confirmación (aknowledgement) y que el emisor tenga que esperar

por ella. Por esta razón Erlang utiliza el método de comunicación "enviar y rezar" (send-and-pray) en el que el emisor no necesita saber si el mensaje fue enviado correctamente y no tiene que esperar respuesta alguna, sigue funcionando.

Por supuesto, no se consigue este nivel de separación entre emisor y receptor porque sí. Copiar la información puede ser muy costoso hablando de estructuras muy grandes y puede causar un alto uso de memoria si el emisor también necesita mantener su propia copia de la información. En la práctica, esto significa que se debe tener cuidado con el manejo del tamaño y la complejidad de los mensajes que se envían. Por lo general, en los programas que usan Erlang, la mayoría de los mensajes que se necesitan enviar son pequeños y la sobrecarga de la copia es generalmente insignificante.

En contraste con otros lenguajes, la concurrencia en Erlang es barata, lanzar (spawn) un proceso cuesta tanto como asignar un objeto en un lenguaje orientado a objetos.

2.7.3 – Operaciones con procesos:

- **Spawning and linking**

Los procesos en Erlang no son hilos de un sistema operativo, son mucho más ligeros y están implementados por el runtime system de Erlang. Erlang es capaz de lanzar fácilmente cientos de miles de procesos en un simple sistema de ejecución y cada uno de esos procesos está separado de todos los demás en tiempo de ejecución, no comparte memoria con los demás y no hay forma de que pueda corromperse por otro proceso.

Un hilo en un sistema operativo modernos reserva algunos megabytes en su dirección de espacio para su pila (lo que significa que una máquina de 32 bit nunca podrá tener mas de unos pocos miles de hilos simultáneos) y aun así se rompe si usa más espacio de lo esperado en la pila. Los procesos en Erlang, por otra parte, empiezan usando sólo un par de cientos de bytes de espacio en la pila cada uno, y este espacio va creciendo o encogiendo automáticamente según se necesite.

La sintaxis para crear un proceso es muy sencilla, en esta línea de código se lanzará un proceso que ejecutará la llamada de función io:format("erlang!") y luego termina:

```
spawn ( io, format, ["erlang!"] ).
```

Y esto es todo, este código empieza un proceso separado que imprime el texto "erlang!" en la consola y luego acaba.

A la hora de lanzar un proceso se puede hacer de dos formas distintas, la primera como hemos visto en el ejemplo anterior sería usando el nombre del módulo (io), el nombre de la función (format) y una lista de argumentos (["erlang"]).

```
Pid = spawn(Module, Function, ListOfArgs).
```

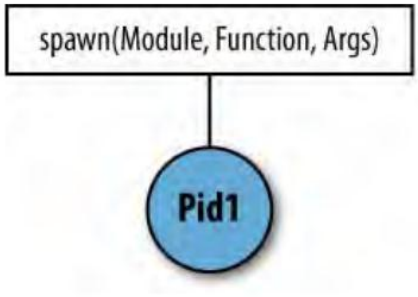


Imagen 8. Spawn/3 de un proceso

La otra manera de lanzar un proceso sería a través de la siguiente estructura donde se utiliza la función "fun" de Erlang como punto de inicio para el nuevo proceso.

```
Pid = spawn(fun() → do_something() end).
```

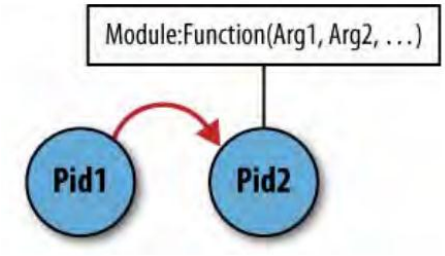


Imagen 9. Spawn a través de otro proceso

También podemos lanzar un proceso haciendo **spawn_link**

```
Pid = spawn_link(...)
```

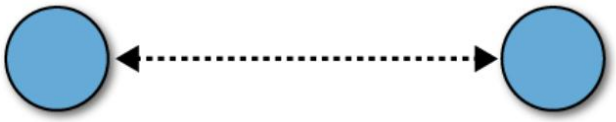


Imagen 10. Spawn_link de un proceso

La diferencia entre lanzar un proceso con **spawn()** y posteriormente hacer **link()** con otro proceso, a usar **spawn_link()**, radica en que con **spawn_link()** el ligar un proceso al nuevo proceso se realiza como una operación atómica, preveniendo un posible comportamiento inesperado entre el tiempo de ligar un proceso a otro.

Todas las funciones **spawn()** devuelven el identificador del nuevo proceso, es por esto que un proceso padre puede comunicarse con su proceso hijo, aunque el proceso hijo no conoce el identificador de su padre a no ser que esta información se le pase desde algún lugar.

Cuando un proceso quiere encontrar su propio identificador de proceso (Pid), puede obternerlo llamando a la función **self()**. Por ejemplo, el siguiente código lanza un proceso hijo que conoce a su proceso padre:

```
Parent = self() ,
Pid = spawn(fun() → myproc:init(Parent) end).
```

Este código especifica con **”myproc:init/1”** donde tendrá el punto de entrada el proceso hijo que queremos lanzar, y como parámetro toma el identificador del padre “**Parent” .**

- **Monitor de proceso:**

Una alternativa a los links es el llamado **monitor**. La figura del monitor consiste en un tipo de link unidireccional que permite a un proceso monitorear a otro sin afectar a su funcionamiento.

```
Ref = monitor(process, Pid).
```

Si el proceso monitoreado identificado con **“Pid”** muere, un mensaje que contiene la referencia única **“Ref”** se envía al proceso que instala el monitor.

- **Tirando una excepción cuando el proceso finaliza:**

La excepción **exit** se usa para terminar un proceso en ejecución. Se usa mediante el BIF (función programada en C que no necesita la declaración de módulo) **exit/1**:

exit (Reason).

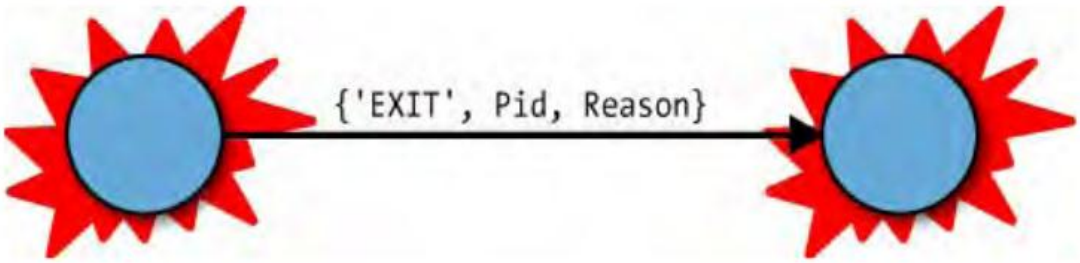


Imagen 11. Tirando una excepción cuando el proceso finaliza

Este código pasará **“Reason”** como parte de su señal de salida (exit signal) a algún otro proceso ligado a este.

- **Enviando un explícito exit signal a un proceso:**

Cuando un proceso muere inesperadamente podemos enviar un exit signal explícito desde el proceso muerto hacia otros. Para que esto ocurra, los procesos no tienen por qué estar ligados.

exit(Pid, Reason).

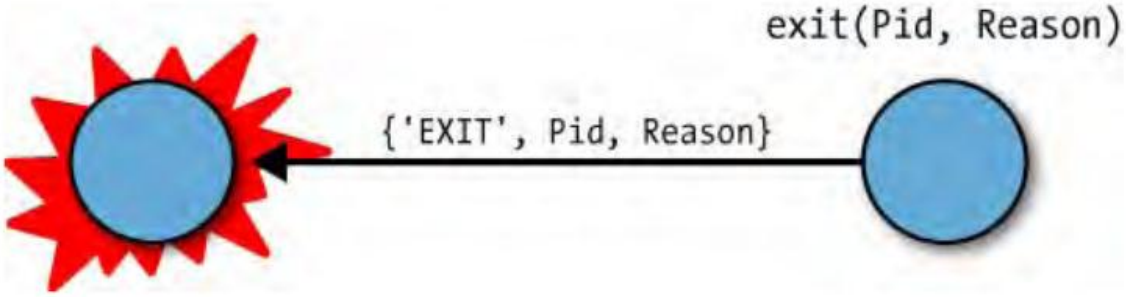


Imagen 12. Exit(Pid, Reason)

Cabe destacar que esta vez se usó el BIF **exit/2** en lugar del **exit/1** , como podemos comprobar, podemos tener dos funciones que se llamen igual y que funcionan completamente diferente, esto es porque Erlang detecta que tienen distinto número de argumentos y tipos de argumentos, de modo que al hacer "**pattern matching"** sabe distinguir a que función nos referimos.

Esta función **"exit/2"** no termina al proceso emisor, pero puede que si al receptor. Si el átomo **"Reason"** es **"kill"** el receptor no podrá atrapar (trap) el exit signal.

- **Configurando la bandera trap_exit:**

Por defecto, un proceso muere si le llega un exit signal de otro proceso ligado. Para prevenir esto, el proceso ligado puede configurar su bander trap_exit:

```
process_flag(trap_exit, true).
```

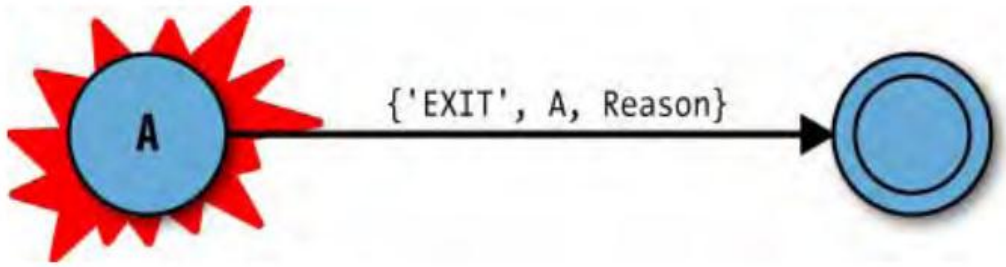


Imagen 13. Trap_exit

De este modo, todos los mensajes entrantes pueden ser atrapados por el trap_exit y no afectarán al proceso. La única excepción como comentamos en el punto anterior sería que le llegase un mensaje **exit (Pid, kill)**.

- **Recibiendo mensajes de forma selectiva:**

Los procesos pueden recibir mensajes y extraerlos de la cola de su buzón de mensajes usando la expresión **"receive"**. Aunque los mensajes entrantes se sitúan en la cola en orden estricto de llegada, el receptor puede decidir que mensaje extraerá de la cola y dejar los otros

en el buzón para después. Esta habilidad selectiva ignora los mensajes irrelevantes, esto es clave en la comunicación entre procesos en Erlang. La forma general de un **"receive"** sería:

```
receive
   Pattern1 when Guard1 → Body1;
   …
   …
   PatternN when GuardN → BodyN
after Time →
   TimeoutBody
end.
```

La parte de **"after Time"** es opcional, si se omite, el **"receive"** nunca dejará de esperar mensajes. Los timeout en Erlang están representados por un número entero de milisegundos o por el átomo **"infinity"**.

El receive empieza a mirar mensajes por la cabeza de la cola, que es el último mensaje que entró, intenta ver que coincidan (match) las cláusulas de los mensajes con los patrones **"Pattern1, …, PatternN"** como si de una espresión case se tratase. Si la cláusula coincide ejecuta el código (siempre y cuando se satisfaga la condición del **"Guard"** de la cláusula) y el mensaje es eliminado del buzón. Si no coincide la cláusula pasa al siguiente mensaje de la cola.

- **Registrar procesos:**

En cada sistema Erlang, hay un proceso local registrado. Un proceso sólo puede tener un identificador único a la vez. Si se llama desde el shell a la función **registered()**, verás en pantalla el siguiente código:

```
1> registered().
[rex, kernel_sup, global_name_server, standard_error_sup,
inet_db, file_server_2, init, code_server, error_logger,
user_drv, application_controller, standard_error,
kernel_safe_sup, global_group, erl_prim_loader, user]
2>
```

Como podemos comprobar, es mucha información. Un sistema Erlang es como un sistema operativo en sí mismo, con un conjunto importante de servicios de sistema ejecutándose. Se puede encontrar el Pid actual registrado bajo un nombre usando la función **whereis().**

```
2>whereis(user).
<0.24.0>
3>
```

"<0.24.0>" es el identificador de proceso del sistema, mientras que **"user"** es el nombre que se le dión a ese identificador, por ese motivo, al ejecutar **"whereis(user)"** la consola nos devuelve su Pid.

También se pueden enviar mensajes directamente a un proceso usando su nombre registrado:

```
1> init ! (stop, stop).
```

En resumen, podemos ver cómo funcionan las siguientes funciones en este código:

```
1>Pid = spawn (timer,  sleep, [60000]).
<0.34.0>
2>register(fred, Pid).
True
3>whereis(fred).
<0.34.0>
4>whereis(fred).                Una vez que pasa el minuto del timer
undefinied
5>
```

- **Comunicación entre procesos:**

Los procesos necesitan hacer algo más que lanzarse y ejecutarse, necesitan intercambiar información, Erlang hace que la comunicación sea simple. El operador básico para enviar un mensaje es "!", se pronuncia "bang" y se usa de la forma "Destino ! Mensaje". Un ejemplo de código de comunicación entre procesos sería:

```
run() →
   Pid = spawn(fun ping/0),
   Pid ! self(),
   receive
        pong → ok
   end.

ping() →
   receive
        From → From ! pong
   end.
```

En este código, la cláusula **"run() ->"** lanza un nuevo proceso que ejectura la función **"ping/0"** , que quiere decir que esa función tiene 0 argumentos. El resultado de lanzar esa función, que es un identificador de proceso se asigna a **"Pid".**

En la siguiente línea **"Pid ! self(),"** se envía a Pid el mensaje "**self()"** , que contiene el identificador del actual proceso que está ejecutándose en la cláusula run.

De este modo, el proceso que se ejecuta tras lanzar ping, recibe en su mailbox ("**receive"**) donde se identifica al emisor ya que **"From"** contiene el identificador del emisor (el proceso que se ejecutó en la cláusula run()) y a este le reenvía el mensaje **"pong"** que entrará en el buzón de mensajes del proceso de la cláusula run(), mensaje que no será ignorado pues coincide con el formato de mensaje, por lo que devolverá como resultado en consola un ok **"pong → ok".**

El framework OTP lleva la comunicación entre procesos a otro nivel en el que más adelante nos sumergiremos a través de algún ejemplo de código.

2.8 – Comportamientos (behaviours).

En este capítulo hablaremos sobre dos de los comportamientos más comunes que adopta Erlang.

La función de un comportamiento en Erlang es la de proveer una plantilla para los procesos de un tipo en particular. Todas las librerías de comportamientos contienen una o más API (generalmente llamadas start o start_link) para lanzar un nuevo contenedor de procesos. A este lanzamiento se le llamar instanciar un comportamiento.

2.8.1 – Comportamiento Gen_Server (Gen_Server Behaviour).

Usamos este comportamiento en el módulo bingo_registry. Este comportamiento se utiliza en los modelos cliente-servidor caracterizado por un servidor central y un arbitrario número de clientes. El modelo cliente-servidor se usa generalmente cuando varios clientes quieren compartir recursos y el servidor es responsable de administrarlos.

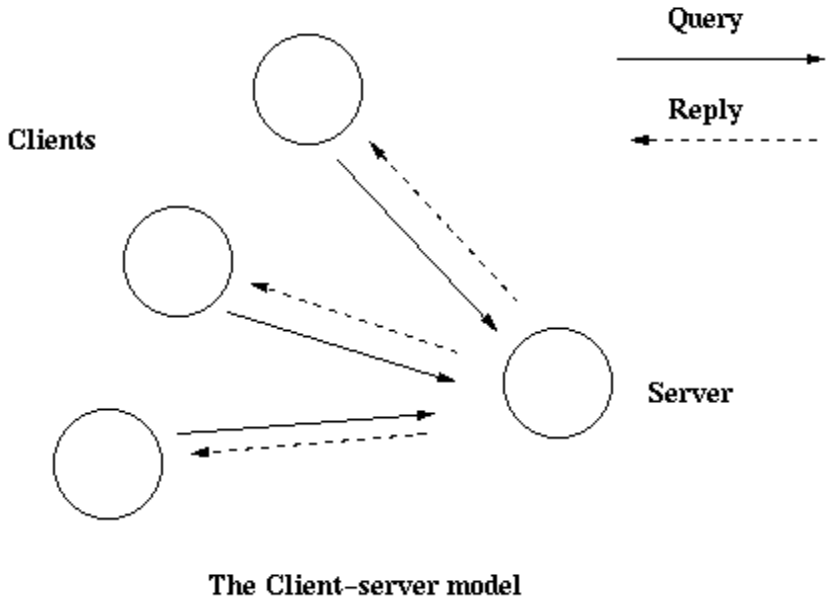


Imagen 14. Gen_server behaviour

Un servidor genérico de procesos (gen_server) está implementado usando un conjunto estándar de funciones (interfaz) e incluye funcionalidad para trazar errores.

Las funciones de un gen_server son las siguientes y se muestran con su relación con sus callbacks:

```
gen_server module              Callback module
-----------------              ---------------
gen_server:start_link -----> Module:init/1

gen_server:call
gen_server:multi_call -----> Module:handle_call/3

gen_server:cast
gen_server:abcast     -----> Module:handle_cast/2

-                     -----> Module:handle_info/2

-                     -----> Module:terminate/2

-                     -----> Module:code_change/3
```

Imagen 15. Callbacks gen_server

Un ejemplo de gen_server implementado sería:

```
-module(ch3).
-behaviour(gen_server).

-export([start_link/0]).
-export([alloc/0, free/1]).
-export([init/1, handle_call/3, handle_cast/2]).

start_link() ->
    gen_server:start_link({local, ch3}, ch3, [], []).

alloc() ->
    gen_server:call(ch3, alloc).

free(Ch) ->
    gen_server:cast(ch3, {free, Ch}).

init(_Args) ->
    {ok, channels()}.

handle_call(alloc, _From, Chs) ->
    {Ch, Chs2} = alloc(Chs),
    {reply, Ch, Chs2}.

handle_cast({free, Ch}, Chs) ->
    Chs2 = free(Ch, Chs),
    {noreply, Chs2}.
```

Imagen 16. Implemantación de un gen_server

2.8.1.2 – Empezando un gen_server:

Como podemos ver en el ejemplo previo, el gen_server empieza con la llamada

```
start_link() ->
    gen_server:start_link({local, ch3}, ch3, [], []) => {ok, Pid}
```

Imagen 17. Lanzando un proceso gen_server

Start_link es la llamada a la función gen_server:start_link/4. Esta función lanza un nuevo proceso ligado, un gen_server.

- El primer argumento `{local, ch3}` especifica el nombre. En este caso, el gen_server estará registrado de manera local como ch3.

  Si el nombre es omitido, el gen_server no será registrado. En su lugar, su pid (identificador de proceso) deberá ser usado. El nombre podría también darse como `{global, Name}`, en este caso el gen_server registrado usaría global:register_name/2.

- El segundo argumento, ch3, es el nombre del callback del módulo.
  En este caso, las funciones de interfaz (start_link, alloc and free) están localizadas en el mismo módulo que las funciones callback (init, handle_call y handle_cast). Esta es normalmente una buena práctica de programación, para así tener el código correspondiente a un proceso contenido en un módulo.

- El tercer argumento, [], es un término el cual se pasa a la función callback init. En este código, init no necesita ninguna información e ignora el argumento.

- El cuarto argumento, [], es una lista de opciones que se le pasan al init.

Si el nombre del registro se realiza con éxito, el nuevo proceso gen_server llamará a la función callback `ch3:init([])` . De init se espera que retorne un `{ok, State}` donde State es el estado interno del gen_server. En este caso, el estado es el disponible channels.

```
init(_Args) ->
    {ok, channels()}.
```

Imagen 18. Arrancando un gen_server

Cabe destacar que gen_server:start_link es síncrono. Esto provoca que no tenga un valor de retorno hasta que el gen_server haya sido inicializado y listo para recibir peticiones.

2.8.1.3 – Peticiones y llamadas síncronas.

Las peticiones síncronas (`alloc()`) se implementan usando gen_server:call/2:

```
alloc() ->
    gen_server:call(ch3, alloc).
```

Imagen 19. Petición síncrona

Ch3 es el nombre del gen_server y debe coincidir con el nombre usado al empezarlo.

La petición se hace mediante un mensaje enviado al gen_server. Cuando la petición se recibe, el gen_server llama al `handle_call(Request, From, State)` que retornará una tupla `{reply, Reply, State1}`. `Reply` es la respuesta que debería ser devuelta al cliente, mientras que State1 es un nuevo valor para el estado del gen_server.

```
handle_call(alloc, _From, Chs) ->
    {Ch, Chs2} = alloc(Chs),
    {reply, Ch, Chs2}.
```

Imagen 20. Llamada síncrona

En este caso, la respuesta está localizada en el canal `ch` y el nuevo estado es el conjunto de canales disponibles `Chs2`.

Así, la llamada `ch3:alloc()` devuelve la localización de canal `ch` y el gen_server entonces espera por nuevas peticiones, ahora con una lista de canales disponibles actualizada.

2.8.1.4 – Peticiones y llamadas asíncronas.

La petición asíncrona `free(Ch)` está implementada usando gen_server:cast/2:

```
free(Ch) ->
    gen_server:cast(ch3, {free, Ch}).
```

Imagen 21. Peticion asíncrona

`ch3` es el nombre del gen_server. `{free, Ch}` es la petición actual.

La petición se hace en un mensaje y se envía al gen_server cast y si es `free` retorna entonces `ok`.

Cuando la petición es recibida, la llamada del gen_server `handle_cast{Request, State}` espera retornar una tupla `{noreply, State1}` donde `State1` es un nuevo valor para el estado del gen_server.

```
handle_cast({free, Ch}, Chs) ->
    Chs2 = free(Ch, Chs),
    {noreply, Chs2}.
```

Imagen 22. Llamada asíncrona

En este caso, el nuevo estado es la lista actualizada de canales disponibles `Chs2`. El gen_server está listo ahora para nuevas peticiones.

2.8.1.5 – Parando un gen_server.

En este apartado explicaremos como se para en caso de ser un gen_server que forma parte de un árbol de supervisión y en el caso de comportarse como un Stand-Alone Gen_server.

- **En un árbol de supervisión**: si el gen_server es parte de un árbol de supervisión, no se necesita ninguna función de parada. El gen_server parará automáticamente por su supervisor. Exactamente se define "shutdown strategy" (estrategia de apagado http://www.erlang.org/doc/design_principles/sup_princ.html#shutdown) cuando se define por un supervisor.

  Si es necesario limpiar todo antes de la finalización, esta estrategia de apagado debe tener un "timeout" y el gen_server debe tener un "trap exit signals" en la función init. Cuando se le ordena apagar, el gen_server podrá ejecutar entonces la función callback terminate(shutdown, State):

```
init(Args) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    {ok, State}.

...

terminate(shutdown, State) ->
    ..code for cleaning up here..
    ok.
```

Imagen 23. Parada gen_server con árbol supervisor

- **Stand_Alone Gen_Servers:** Si el gen_server no forma parte de un árbol de supervisión, la función de stop puede ser útil, por ejemplo:

```
...
export([stop/0]).
...

stop() ->
    gen_server:cast(ch3, stop).
...

handle_cast(stop, State) ->
    {stop, normal, State};
handle_cast({free, Ch}, State) ->
    ....

...

terminate(normal, State) ->
    ok.
```

Imagen 24. Parada gen_server de tipo Stand-Alone

La función callback que maneja la petición de stop, devuelve una tupla `{stop, normal, State1}`, donde normal especifica que se paró de forma normal y `State1` es un nuevo valor para el estado del gen_server. Esto provocará que el gen_server pueda llamar a `terminate(normal, State1)` y así acabar de forma satisfactoria.

2.8.1.6 – Manejando otros mensajes.

El gen_server debería ser capaz de recibir otros mensajes que no son peticiones, la función callback handle_info (Info, State) debe ser implementada para manejarlos. A continuación mostramos ejemplos de otros mensajes y mensajes de salida, cuando el gen_server está ligado a otro proceso (como el supervisor) y atrapando señales de salida.

```
handle_info({'EXIT', Pid, Reason}, State) ->
    ..code to handle exits here..
    {noreply, State1}.
```

Imagen 25. Manejo de mensajes que no son peticiones

El método code_change también tiene que ser implementado:

```
code_change(OldVsn, State, Extra) ->
    ..code to convert state (and more) during code change
    {ok, NewState}.
```

Imagen 26. Método para actualizar el sistema

2.8.2 – Comportamiento Gen_FSM (Finite State Machine).

Este comportamiento implementa para un módulo una máquina de estados finita. Un proceso de una máquina de estados finita se implementa usando este módulo que tendrá un interfaz de funciones.

Igual que al gen_server lo forman las siguientes llamadas:

```
gen_fsm module                  Callback module
--------------                  ---------------
gen_fsm:start_link              -----> Module:init/1

gen_fsm:send_event              -----> Module:StateName/2

gen_fsm:send_all_state_event    -----> Module:handle_event/3

gen_fsm:sync_send_event         -----> Module:StateName/3

gen_fsm:sync_send_all_state_event -----> Module:handle_sync_event/4

-                               -----> Module:handle_info/3

-                               -----> Module:terminate/3

-                               -----> Module:code_change/4
```

Imagen 27. Callbacks gen_fsm

Una máquina de estados finita podría describirse como un conjunto de relaciones con de la forma:

```
State (S) x Event (E) -> Actions (A), State(S')
```

Estas relaciones son interpretadas como:
Si estamos en un estado S y el evento E ocurre, deberíamos realizas la acción A y hacer una transición al estado S'.

Para una máquina FSM implementada usando el comportamiento gen_fsm, las reglas de transición de estado están escritas como un número de funciones Erlang las cuales conforman lo siguiente:

```
StateName(Event, StateData) ->
    .. code for actions here ...
    {next_state, StateName', StateData'}
```

Imagen 28. Reglas de transición de la máquina de estados

A continuación vamos a ver un ejemplo de la implementación de máquina de estados finita.
En este caso, tenemos una puerta con un candado. Inicialmente, la puerta está cerrada con el candado. Una vez que alguien presione un botón, se genera un evento. Dependiendo de qué botones se hayan presionado antes, la secuencia que abra el candado puede que sea la correcta, incompleta o incorrecta.

Si es la correcta, la puerta se abrirá durante 30 segundos (30000 milisegundos). Si la secuencia es incompleta, esperaremos a que otro botón sea presionado. Si es incorrecta, empezaremos de nuevo la secuencia.

```
-module(code_lock).
-behaviour(gen_fsm).

-export([start_link/1]).
-export([button/1]).
-export([init/1, locked/2, open/2]).

start_link(Code) ->
    gen_fsm:start_link({local, code_lock}, code_lock, lists:reverse(Code

button(Digit) ->
    gen_fsm:send_event(code_lock, {button, Digit}).

init(Code) ->
    {ok, locked, {[], Code}}.

locked({button, Digit}, {SoFar, Code}) ->
    case [Digit|SoFar] of
        Code ->
            do_unlock(),
            {next_state, open, {[], Code}, 30000};
        Incomplete when length(Incomplete)<length(Code) ->
            {next_state, locked, {Incomplete, Code}};
        _Wrong ->
            {next_state, locked, {[], Code}}
    end.

open(timeout, State) ->
    do_lock(),
    {next state, locked, State}.
```

Imagen 29. Implementación de un gen_fsm

2.8.2.1 – Empezando un Gen_Fsm.

En el ejemplo anterior, el gen_fsm es empezado por la llamada `code_lock:start_link(Code)`:

```
start_link(Code) ->
    gen_fsm:start_link({local, code_lock}, code_lock, lists:reverse(Code
```

Imagen 30. Lanzar un proceso gen_fsm

Start_link llama a la función gen_fsm:start_link/4. Esta función se lanza y se liga a un nuevo proceso, un gen_fsm.

- El primer argumento `{local, code_lock}` especifica el nombre. En este caso, el gen_fsm se registrará como code_lock.
-

  Si el nombre se omite, el gen_fsm no estará registrado. En su lugar su pid (número de identificación de proceso) debe ser usado. También se le podría dar el nombre de `{global, Name}`, en ese caso el gen_fsm es registrado usando global:register_name/2.

- El segundo argumento, code_block, es el nombre del módulo callback, que es el módulo donde se encuentra la función callback.

En este caso, las funciones del interfaz (start_link and button) están situadas en el mismo módulo que las funciones callback (init, locked and open). Esto normalmente es una buena práctica de programación, se recomienda tener el código correspondiente a un proceso en el mismo módulo.

- El tercer argumento, `Code`, es una lista de dígitos la cual se pasa invertida a la función callback init. Es aquí, en el `init` donde se tiene el código correcto para desbloquear la puerta.

- El cuarto argumento, `[]`, es una lista de opciones.

Si el registro se realiza con éxito, el nuevo proceso gen_fsm llama a la función callback `code_lock:init (Code)`. Esta función esperará un retorno `{ok, StateName, StateData}` donde `StateName` es el nombre del estado inicial del gen_fsm. En este caso `locked` (cerrado), asumiendo que la puerta está cerrada al empezar el ejercicio. `StateData` es el estado interno del gen_fsm. En este caso, el state data es la secuencia del botón con el código correcto para abrir la puerta.

```
init(Code) ->
    {ok, locked, {[], Code}}.
```

Imagen 31. Arrancando un gen_fsm

Cabe destacar que gen_fsm:start_link es síncrono. Esto hace que no tenga un valor de retorno hasta que el gen_fsm esté inicializado y listo para recibir notificaciones.

2.8.2.2 – Notificar sobre los eventos.

La función de notificación es gen_fsm:send_event/2:

```
button(Digit) ->
    gen_fsm:send_event(code_lock, {button, Digit}).
```

Imagen 32. Notificación de evento

`Code_lock` es el nombre del gen_fsm y debe coincidir con el nombre usado para empezarlo. `{button, Digit}` es el evento actual.

El evento tiene lugar mediante un mensaje enviado al gen_fsm. Cuando el evento es recibido, el gen_fsm llama `StateName(Event, StateData)` el cual esperará el retorno de la tupla `{next_state, StateName1, StateData1}`. `StateName` es el nombre del actual estado y `StateName1` es el nombre del siguiente estado al que irá. `StateData1` es el nuevo valor para el state data del gen_fsm.

```
locked({button, Digit}, {SoFar, Code}) ->
    case [Digit|SoFar] of
        Code ->
            do_unlock(),
            {next_state, open, {[], Code}, 30000};
        Incomplete when length(Incomplete)<length(Code) ->
            {next_state, locked, {Incomplete, Code}};
        _Wrong ->
            {next_state, locked, {[], Code}};
    end.

open(timeout, State) ->
    do_lock(),
    {next_state, locked, State}.
```

Imagen 33. Transición de estado

Si la puerta está cerrada y el botón es presionado, la secuencia completa del botón se comparará con la secuencia correcta del candado y dependiendo del resultado, la puerta se abrirá y el gen_fsm se pondrá en el estado open, o la puerta se mantendrá cerrada.

2.8.2.3 – Todos los eventos de estado.

Algunas veces un evento puede llegar en algún estado del gen_fsm. En su lugar de enviar el mensaje con gen_fsm:send_event/2 y escribir una cláusula manejando el evento para cada situación, el mensaje puede ser enviado con gen_fsm:send_all_state_event/2 y manejado con Module:handle_event/3:

```
-module(code_lock).
...
-export([stop/0]).
...

stop() ->
    gen_fsm:send_all_state_event(code_lock, stop).

...

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.
```

Imagen 34. Manejar eventos en cualquier estado

2.8.2.4 – Parando un gen_fsm.

Explicaremos cómo se para un gen_fsm en caso de formar parte de un árbol de supervisión, o en caso de ser un gen_fsm Stand-Alone.

- **Formando parte de un árbol de supervisión:** si un gen_fsm es parte de un árbol de supervisión no se necesita una función de parado, el gen_fsm parará automáticamente por el supervisor. Debe pararse exactamente como es definida por una shutdown

strategy (http://www.erlang.org/doc/design_principles/sup_princ.html#shutdown) en el supervisor.

Si es necesario limpiar la información antes de parar, la shutdown strategy debe tener un timeout y un gen_fsm con un conjunto de señales de atrapado de salida en la función init. Cuando se ordena apagar, el gen_fsm deberá llamar a la función callback terminate (shutdown, StateName, StateData):

```
init(Args) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    {ok, StateName, StateData}.

...

terminate(shutdown, StateName, StateData) ->
    ..code for cleaning up here..
    ok.
```

Imagen 35. Parar gen_fsm formando parte de un árbol supervisor

- **Stand-Alone Gen_Fsm:** si el gen_fsm no forma parte de un árbol de supervisión, una función stop puede ser útil, por ejemplo:

```
...
-export([stop/0]).
...

stop() ->
    gen_fsm:send_all_state_event(code_lock, stop).
...

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.

...

terminate(normal, _StateName, _StateData) ->
    ok.
```

Imagen 36. Parar gen_fsm Stand-Alone

La función callback que maneja el evento stop retorna una tupla `{stop, normal, StateData1}`, donde `normal` especifica que es una parada normal y `StateData1` es un nuevo valor para la información del estado del gen_fsm. Esto provoca al gen_fsm a llamar a `terminate(normal, StateName, StateData1)` y entonces terminar de forma exitosa.

2.8.2.5 – Manejando otros mensajes.

El gen_fsm debería ser capaz de recibir otros mensajes que no sean eventos, la función callback `handle_info(Info, StateName, StateData)` debe ser implementada para manejarlos.

```
handle_info({'EXIT', Pid, Reason}, StateName, StateData) ->
    ..code to handle exits here..
    {next state, StateName1, StateData1}.
```

Imagen 37. Manejando mensajes sin eventos

El método `code_change` también tiene que ser implementado.

```
code_change(OldVsn, StateName, StateData, Extra) ->
    ..code to convert state (and more) during code change
    {ok, NextStateName, NewStateData}
```

Imagen 38. Actualizar codigo gen_fsm

2.9 – Buenas prácticas.

En este capítulo se listarán algunos aspectos que deberían ser tenidos en consideración a la hora de desarrollar software usando Erlang.

- **Estructura y terminología Erlang:**

Los sistemas Erlang están divididos en módulos. Los módulos están compuestos de funciones y atributos. A las funciones de un módulo sólo se les puede invocar desde ese mismo módulo a no ser que sean exportadas, en ese caso, podrían ser invocadas desde otros módulos. Los atributos comienzan con "-" y se sitúan en el inicio de un módulo.

El trabajo de un sistema diseñado usando Erlang lo realizan los procesos. Un proceso es un trabajador el cual puede usar funciones en varios módulos. Los procesos pueden comunicarse entre ellos enviándose mensajes. Los procesos pueden recibir mensajes enviados por otros procesos.

Un proceso puede supervisar la existencia de otro proceso mediante un link (procesos ligados). Cuando un proceso termina, automáticamente envía un exit signal al proceso al que está ligado. Un proceso puede cambiar su comportamiento por defecto a través de "trapping exits", esto provoca que todos los "exit signals" enviados a un proceso se vuelvan mensajes.

Una función pura es una función que retorna con los mismos argumentos dados el mismo valor sin importar el contexto de la llamada de la función. Esto es lo que se espera normalmente de una función matemática. Una función no pura se dice que tiene efectos secundarios.

- **Principios ingeniería del software:**

  - **Cuantas menos funciones se exporten de un módulo mejor.**

Los módulos son la entidad básica de código estructurado en Erlang. Un módulo puede contener un gran número de funciones pero sólo funciones incluidas en la lista "export" del módulo pueden ser llamadas desde fuera del módulo. De esta manera, es más sencillo cambiar el código de un módulo sin cambiar la estructura interna de este si se exportan pocas funciones.

  - **Intentar reducir las dependencias entre módulos.**

Un módulo que llama a funciones de varios módulos diferentes será más difícil de mantener que un módulo que sólo llama a funciones de pocos módulos distintos.

Esto ocurre porque cada vez que hacemos un cambio en el interfaz de un módulo, tenemos que comprobar todos los bloques de código donde se llama a funciones de otros módulos. Reduciendo las interdependencias entre módulos simplifica el problema de mantenimiento de estos.

Se desea que las llamadas entre módulos sigan una estructura de árbol.

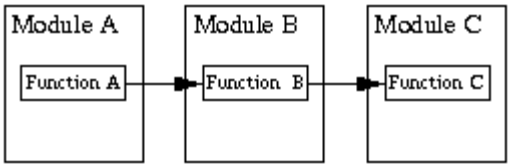


Imagen 39. Estructura de arbol

Y no se desea que las llamadas entre módulos hagan ciclos.

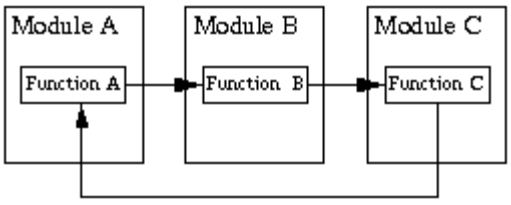


Imagen 40. Estructura con ciclos

  - **Usar código comúnmente usado en las librerías.**

Normalmente el código usado debería estar en librerías. Las librerías son colecciones de funciones. Así, una librería como "lists" contiene exclusivamente funciones para manipular listas. Las librerías nos permite reusar código de una manera segura.

- **Aislar código “dirty” (sucio) o “tricky” (complicado) separado en módulos.**

A menudo un problema puede ser resuelto usando una mezcla de código sucio y limpio. En este caso deberíamos separar el código limpio del sucio en módulos diferentes.

Llamamos código sucio a aquel código que realiza trabajos sucios como por ejemplo:

- Usar el diccionario de procesos.

- Usar erlang:process_info/1 para extraños propósitos.

- Hacer algo que se supone que no debes hacer, pero que tienes que hacer.

- **No asumir lo que una llamada va a hacer con los resultados de una función.**

Por ejemplo, supongamos que llamamos a una rutina con ciertos argumentos que deben ser inválidos. El ejecutor de la rutina no debe hacer alguna suposición sobre lo que la invocación de la función espera que ocurra cuando los argumentos son inválidos. De este modo no deberíamos escribir código como en la siguiente imagen:

```
do_something(Args) ->
  case check_args(Args) of
    ok ->
      {ok, do_it(Args)};
    {error, What} ->
      String = format_the_error(What),
      io:format("* error:~s\n", [String]), %% Don't do this
      error
  end.
```

En su lugar deberíamos escribir algo como en la siguiente imagen:

```
do_something(Args) ->
  case check_args(Args) of
    ok ->
      {ok, do_it(Args)};
    {error, What} ->
      {error, What}
  end.

error_report({error, What}) ->
  format_the_error(What).
```

En el primer caso, la cadena **“error”** siempre se imprime como una salida estándar, en el segundo caso el error se devuelve a la aplicación. La aplicación puede ahora decidir qué hacer con el error devuelto.

- **Abstraer los patrones comunes de código o comportamiento.**

Siempre que se tenga el mismo patrón de código en dos o más lugares del código hay que intentar encapsularlo en una función común, de este modo se podrá llamar a la función en lugar de tener el código repetido en distintos sitios.

- **Top-down.**

Escribe tu programa usando el estilo "Top-down". Es una buena manera de acercarse a los detalles de implementación sucesivamente, acabando por definir las funciones primitivas. El código será independiente y más fácil de interpretar que con un desconocido código de alto nivel implementado.

- **No optimizar el código.**

No optimizar el código en primera instancia. Primero se aconseja hacer que funcione bien y después, si es necesario, que funcione bien y más rápido.

- **Usar el principio de "least astonishment" (menos asombro).**

El sistema debe responder de una manera que no asombre al usuario, es mejor usar código predecible que ocurra cuando el usuario haga algo obteniendo un resultado que no sorprenda.

- **Intentar eliminar efectos secundarios.**

Erlang tiene algunas primitivas con efectos secundarios. Estas funciones no pueden ser reusadas fácilmente ya que causan cambios en el contexto y tienes que tener el estado exacto del proceso que tenías antes de llamar a estas rutinas.

  - Escribe tanto código libre de efectos secundarios como sea posible.
  - Intenta maximizar el número de funciones puras.
  - Junta las funciones que tienen efectos secundarios y documéntalas de forma clara y precisa.
  - Escribiendo el código con cuidado es posible obtener un código libre de efectos secundarios, lo que hará que el sistema sea mucho más fácil de mantener, probar y entender.

- **Hacer código lo más determinista posible.**

Un programa determinista es aquel que siempre se ejecuta de la misma manera sin importar cuantas veces se ejecuta. Un programa no determinista puede obtener distintos resultados cada vez que se ejecuta. Para los propósitos de depurar es una buena idea hacer las cosas los más deterministas posible.

- **Aislar interfaces hardware con drivers de dispositivos.**

El hardware debe estar aislado del sistema aunque se usen drivers de los dispositivos. Los drivers deben implementar interfaces hardware los cuales hacen al hardware aparecer como si fueran procesos Erlang. El hardware debe estar hecho para observar el comportamiento como un proceso normal Erlang. Debería recibir y enviar mensajes Erlang y debería responder con normalidad a los distintos errores que pueden ocurrir.

- **Hacer o deshacer cosas en la misma función.**

Supongamos que tenemos un programa que abre un archivo, hace algo con el y se cierra después. Esto debe estar codificado como en la siguiente imagen:

```
do_something_with(File) ->
  case file:open(File, read) of,
    {ok, Stream} ->
      doit(Stream),
      file:close(Stream) % The correct solution
    Error -> Error
  end.
```

Observemos la simetría al abrir un archivo **“file:open”** and al cerrarlo **“file:close"** en la misma rutina. La solución abajo es más difícil de seguir y no es tan obvia. No debe programarse del siguiente modo como se muestra en la siguiente imagen:

```
do_something_with(File) ->
  case file:open(File, read) of,
    {ok, Stream} ->
      doit(Stream)
    Error -> Error
  end.

doit(Stream) ->
  ....,
  func234(...,Stream,...).
  ...

func234(..., Stream, ...) ->
  ...,
  file:close(Stream) %% Don't do this
```

- **Control de errores:**

- **Separar el código normal del manejo de errores:**

No debe saturarse el código normal con el código diseñado para el manejo de errores. Debe impedirse en la medida de lo posible tener código exclusivamente para “caso normal”. Si este código fallase, el proceso debería reportar el error y estallar tan pronto como es posible. Sin intentar de arreglar el error y continuar. El error debe ser manejado en un proceso diferente.

Separar los dos tipos de código hace mejor y más fácil de mantener un sistema.

- **Identificar el error de kernel:**

Uno de los elementos básicos del diseño de un sistema es identificar cual es la parte del sistema que tiene que ser correcta y cual no tiene que serlo.

En los diseños de los sistemas operativos convencionales el kernel del sistema asume serlo y debe ser correcto, mientras todas las aplicaciones de usuario no tienen por qué ser

correctas necesariamente. Si una aplicación de usuario falla sólo le concierne a la aplicación donde se produce el fallo, sin afectar a la integridad del conjunto del sistema.

El primer paso del diseño de un sistema debe ser identificar que parte del sistema debe ser correcto, a esta parte del sistema le llamamos "error kernel". A menudo el "error kernel· tiene algún tipo de memoria en tiempo real residente en la base de datos que almacena los estados del software.

- **Procesos, servidores y mensajes.**

- **Implementar un proceso en un módulo:**

El código para implementar un sencillo proceso debe estar contenido en un módulo. Un proceso puede llamar a funciones en algunas librerías pero el código para el "top loop" del proceso debe estar contenido en un sólo módulo. El código para el "top loop" de un proceso no debe estar separado en varios módulos, esto haría que el control de flujo fuese extremadamente complicado a la hora de entender.

- **Usar procesos para estructurar el sistema:**

No debe usarse los procesos y el paso de mensajes cuando una función puede ser usada en su lugar.

- **Registro de procesos:**

Los procesos registrados deben ser registrados con el mismo nombre que el módulo. Esto hace más sencillo encontrar el código para un proceso.

Solo los procesos registrados deben tener un largo ciclo de vida.

- **Asignar exactamente un proceso paralelo a cada una de la actividades concurrente verdaderas en el sistema:**

A la hora de decidir si implementar algo mediante procesos secuenciales o paralelos entonces se debe utilizar la estructura implícita en la estructura intrínseca del problema. La regla principal es:

"Utilice un proceso paralelo para modelar cada actividad verdaderamente concurrente en el mundo real".

Si existe un "mapping" uno a uno entre el número de procesos paralelos y el número de verdaderas actividades paralelas en el mundo real, el programa será más sencillo de entender.

- **Cada proceso debe tener un rol:**

Los procesos tienen diferentes roles in el sistema, por ejemplo en el modelo cliente-servidor.

Un proceso debe tener un rol en la medida de lo posible, por ejemplo, un proceso puede ser un cliente o un servidor pero no debe combinar estos roles.

Otros roles que puede asumir un proceso son los de:

- Supervisor: vigila otros procesos hijos y los reinicia si fallan.
- Worker: un proceso normal que puede fallar.
- Trusted worker: un proceso al que no se le permite tener errores.

- **Usar funciones genéricas para servidores y controladores de protocolo siempre que sea posible:**

En algunas circunstancias es una buena idea usar servidores genéricos como el **"generic server"** implementado en las librerías estándar. Simplificará el uso y mejorará la estructura del sistema.
Se aplica el mismo argumento para la mayoría de software de control de protocolo en el sistema.

- **Etiquetar mensajes:**

Todos los mensajes deben ser etiquetados. Esto hace que se puedan implementar nuevos mensajes con mayor facilidad.

Una de las razones de escoger distintas etiquetas es la de facilitar al depuración.

Una mala solución se muestra en la siguiente imagen:

```
loop(State) ->
  receive
    ...
    {Mod, Funcs, Args} -> % Don't do this
      apply(Mod, Funcs, Args},
      loop(State);
    ...
  end.
```

En la siguiente imagen se etiquetan correctamente los mensajes:

```
loop(State) ->
  receive
    ...
    {execute, Mod, Funcs, Args} -> % Use a tagged message.
      apply(Mod, Funcs, Args},
      loop(State);
    {get_status_info, From, Option} ->
      From ! {status_info, get_status_info(Option, State)},
      loop(State);
    ...
  end.
```

- **Desechar mensajes desconocidos:**

Todos los servidores deben tener al menos un **“receive”** alternativo. Esto sirve para evitar las colas de mensajes.

```
main_loop() ->
  receive
    {msg1, Msg1} ->
      ...,
      main_loop();
    {msg2, Msg2} ->
      ...,
      main_loop();
    Other -> % Flushes the message queue.
      error_logger:error_msg(
          "Error: Process ~w got unknown msg ~w~n.",
          [self(), Other]),
      main_loop()
  end.
```

- **Escribir servidores “tail-recursive”:**

Todos los servidores deben ser **“tail-recursive”** , de lo contrario el sistema consumirá memoria hasta que se quede sin ella. Debemos evitar escribir el siguiente código:

```
loop() ->
  receive
    {msg1, Msg1} ->
      ...,
      loop();
    stop ->
      true;
    Other ->
      error_logger:log({error, {process_got_other, self(), Other}}),
      loop()
  end,
  io:format("Server going down").                % Don't do this!
               % This is NOT tail-recursive
```

En su lugar debemos escribir:

```
loop() ->
  receive
    {msg1, Msg1} ->
      ...,
      loop();
    stop ->
      io:format("Server going down");
    Other ->
      error_logger:log({error, {process_got_other, self(), Other}}),
      loop()
  end. % This is tail-recursive
```

- **Interfaz de funciones:**

Usar funciones para interfaces siempre que sea posible, evitar enviar mensajes directamente. Encapsular el paso de mensajes en las funciones del interfaz. Hay casos en los que no se puede hacer esto.

El protocolo de mensaje es la información interna y se debe ocultar a otros módulos.

Un ejemplo de funciones de una interfaz sería:

```
-module(fileserver).
-export([start/0, stop/0, open_file/1, ...]).

open_file(FileName) ->
  fileserver ! {open_file_request, FileName},
  receive
    {open_file_response, Result} -> Result
  end.

...<code>...
```

- **"Time-outs":**

Hay que tener cuidado cuando se usa **"after"** en el **"receive"**. Hay que estar seguros de que se maneja el caso cuando el mensaje llega más tarde (ver **"Desechar mensajes desconocidos"**).

- **"Trapping exits":**

Los procesos deben ser capaces de atrapar **"exit signals"** o no, es una mala práctica hacer que un proceso alterne el poder atrapar estas señales o no.

- **Varios convenios específicos de Erlang:**

- **Usar registros como principal estructura de datos:**

Utilizar registros como principal estructura de datos. Un registro es una tupla etiquetada y se introdujo en Erlang en la versión 4.3. Es similar al **"struct"** en C o **"record"** en Pascal.

Si el registro se usa en varios módulos, su definición debe situarse en la cabecera del archivo del módulo. Si el registro sólo se usa dentro de un módulo, la definición del registro debe estar en el inicio del archivo donde el módulo está definido.

Las características del registro en Erlang pueden utilizarse para asegurar la consistencia del módulo de estructura de datos y por lo tanto deben ser usados por las funciones de interfaz al pasar estructuras de datos entre los módulos.

- **Usar selectores y constructores:**

Utilizar selectores y constructores que proporciona el registro para manejar instancias de registro. No debe utilizarse "matching" ya que se asume que el registro es una tupla. Ejemplo:

```
demo() ->
  P = #person{name = "Joe", age = 29},
  #person{name = Name1} = P,% Use matching, or...
  Name2 = P#person.name. % use the selector like this.
```

Y no se debe escribir código como:

```
demo() ->
  P = #person{name = "Joe", age = 29},
  {person, Name, _Age, _Phone, _Misc} = P. % Don't do this
```

- **Usar etiquetas en los valores devueltos:**

Usar etiquetas en los valores que se retornan. No debe programarse como:

```
keysearch(Key, [{Key, Value}|_Tail]) ->
  Value; %% Don't return untagged values!
keysearch(Key, [{_WrongKey, _WrongValue} | Tail]) ->
  keysearch(Key, Tail);
keysearch(Key, []) ->
  false.
```

Entonces la tupla **"{Key, Value}"** no puede contener el valor falso. Aquí se corrige el problema:

```
keysearch(Key, [{Key, Value}|_Tail]) ->
  {value, Value}; %% Correct. Return a tagged value.
keysearch(Key, [{_WrongKey, _WrongValue} | Tail]) ->
  keysearch(Key, Tail);
keysearch(Key, []) ->
  false.
```

- **Usar catch and throw con extremo cuidado:**

No debe utilizarse **catch** and **throw** a menos que sepas exactamente lo que estás haciendo. Debe usarse las menos veces posibles.

Catch and **throw** pueden ser muy útiles cuando el programa controle entradas complicadas y difíciles de leer, la cual puede causar errores en varios lugares dentro del código. Un ejemplo de esto es un compilador.

- **Usar el diccionario de procesos con extremo cuidado:**

No debe usarse **get, put,** etc... a menos que sepas exactamente lo que estás haciendo. Debe usarse las menos veces posibles.

Una función que usa el diccionario de procesos puede reescribirse introduciendo un nuevo argumento. Un ejemplo de cómo no debe hacerse sería:

```
tokenize([H|T]) ->
  ...;
tokenize([]) ->
  case get_characters_from_device(get(device)) of % Don't use get/1!
    eof -> [];
    {value, Chars} ->
      tokenize(Chars)
  end.
```

La solución correcta sería:

```
tokenize(_Device, [H|T]) ->
  ...;
tokenize(Device, []) ->
  case get_characters_from_device(Device) of     % This is better
    eof -> [];
    {value, Chars} ->
      tokenize(Device, Chars)
  end.
```

El uso de **get** o **put** provocará que la función se comporte de manera diferente cada vez que se llame con la misma entrada en distintas ocasiones. Esto hace que el código no sea determinista y lo hace más difícil de leer.

- **No usar import en la medida de lo posible:**

No debe utilizarse **“-import”**, usándolo hace que el código sea más difícil de leer y no se puede ver directamente en qué módulo se define una función. Puede usarse **“exref”** (**Cross Reference Tool)** para encontrar las dependencias del módulo.

- **Exportar funciones:**

Se detallarán los motivos por la que una función debe ser exportada.

- Es un interfaz de usuario en el módulo.
- Es un interfaz de funciones para otros módulos.
- Es invocada por “apply”, **“spawn”**, etc... pero solo desde dentro de su módulo.

Deben escribirse distintos grupos de **-export** y comentarlos como se muestra a continuación:

```
%% user interface
-export([help/0, start/0, stop/0, info/1]).

%% intermodule exports
-export([make_pid/1, make_pid/3]).
-export([process_abbrevs/0, print_info/5]).

%% exports for use within module only
-export([init/1, info_log_impl/1]).
```

- **Léxico específico y convenios de estilo.**

- **No escribir código extremadamente anidado:**

El código anidado es aquel que contiene **case/if/receive** dentro de otras **case/if/receive**. Esto es un mal estilo de programación.

- **No escribir módulos muy grandes:**

Un módulo no debe contener más de 400 líneas de código fuente. Es mejor tener varios módulos pequeños.

- **No escribir funciones muy grandes:**

No se debe escribir una función que contenga más de 15 o 20 líneas de código. Dividir una función grande en varias pequeñas es una buena práctica de programación. Un problema no se va a resolver por escribir una función más grande.

- **No escribir líneas muy largas:**

No se deben escribir líneas muy largas. Una línea no debe tener más de 80 caracteres.

- **Nombres de variables:**

Escoger nombres de variables significativos, es más difícil de lo que parece.

Si el nombre de una variable consiste en unas pocas palabras, debe usarse **"_"** o poner la letra inicial de la siguiente palabra en mayúscula. Por ejemplo : **My_variable** or **MyVariable.**

Si no te importa el valor de la variable en un momento dado, pero quieres ponerla en algún lugar como una tupla para hacer "matching" puedes usar la barra baja al inicio de la variable **_MyVariable.**

- **Nombres de funciones:**

El nombre de una función debe referirse exactamente a lo que la función hace.

Las funciones en diferentes módulos que resuelven los mismos problemas deben tener el mismo nombre. Un mal nombre para una función es aquel de los errores más comunes en programación.

Algún tipo de convenios a la hora de nombrar las funciones son muy útiles por ejemplo, el prefijo **"is_"** puede utilizarse dar el significado de que la función en cuestión retorna el átomo **true** o **false.**

```
is_...() -> true | false
check_...() -> {ok, ...} | {error, ...}
```

- **Nombres de módulos:**

Erlang tiene una estructura de módulos plana, sin embargo, a menudo podemos obtener un efecto parecido a un modelo de estructura jerárquica. Esto puede hacerse con conjuntos de módulos relacionados nombrados con el mismo prefijo.

Si, por ejemplo, un controlador ISDN es implementado usando cinco distintos módulos relacionados, estos módulos deberían llamarse como:

```
isdn_init
isdn_partb
isdn_...
```

- **Documentando código:**

- **Atribuir código:**

Debe atribuirse correctamente el código en la cabecera del módulo. Decir con qué ideas se ha contribuido o de dónde proviene el módulo.

Nunca debe robarse código. Ejemplos de atribuciones útiles son:

```
-revision('Revision: 1.14 ').
-created('Date: 1995/01/01 11:21:11 ').
-created_by('eklas@erlang').
-modified('Date: 1995/01/05 13:04:07 ').
-modified_by('mbj@erlang').
```

- **Proporcionar referencias en el código para las especificaciones:**

Proporcionar referencias a través del código a algunos documentos relevantes que ayuden a entender el código.

- **Documentar todos los errores:**

Todos los errores deben ser listados juntos con una descripción en inglés y separados del código en un documento.

Obviamente se refiere a errores del sistema.

- **Documentar todas las estructuras de datos principales en mensajes:**

Usar tuplas etiquetadas como principal estructura de datos a la hora de enviar mensajes entre distintas partes del sistema. Como característica de los registros destaca que pueden ser usados para asegurar la consistencia a través del módulo de estructura de datos.

- **Comentarios:**

Los comentarios deben ser claros y concisos evitando palabras sin significado. Los comentarios deben estar escritos en inglés.

Los comentarios sobre el módulo deben empezar con tres caracteres de porcentaje **(%%%)**.

Los comentarios sobre una función deben empezar con dos caracteres de porcentaje **(%%).**

Los comentarios dentro del código Erlang deben comenzar por un carácter de porcentaje **(%).**

```
%% Comment about function
some_useful_functions(UsefulArgugument) ->
  another_functions(UsefulArgugument),    % Comment at end of line
  % Comment about complicated_stmnt at the same level of indentation
  complicated_stmnt,
```

- **Comentar cada función:**

Lo más importante a la hora de comentar una función es:

  - El propósito de la función.
  - El dominio de las entradas válidas a la función.
  - El dominio de las salidas de la función.
  - Si la función implementa un algoritmo complicado debe describirse.
  - Las causas posibles de que se produzca un fallo así como sus “exit signals” generadas por **“exit/1”, “throw/1”** o algún error no esperado en tiempo de ejecución. Cabe destacar la diferencia entre un fallo y un error retornado.
  - Algún efecto secundario de la función en caso de tenerlo.

Ejemplo:

```
%%--------------------------------------------------------------------
%% Function: get_server_statistics/2
%% Purpose: Get various information from a process.
%% Args:   Option is normal|all.
%% Returns: A list of {Key, Value}
%%     or {error, Reason} (if the process is dead)
%%--------------------------------------------------------------------
get_server_statistics(Option, Pid) when pid(Pid) ->
```

- **Estructura de datos:**

El registro debe definirse junto con un texto plano que lo describa. Por ejemplo:

```
%% File: my_data_structures.h

%%--------------------------------------------------------------------
%% Data Type: person
%% where:
%%    name: A string (default is undefined).
%%    age: An integer (default is undefined).
%%    phone: A list of integers (default is []).
%%    dict:     A dictionary containing various information about the person.
%%       A {Key, Value} list (default is the empty list).
%%--------------------------------------------------------------------
-record(person, {name, age, phone = [], dict = []}).
```

- **Cabeceras de archivos, copyright:**

Cada archivo del código fuente debe empezar con la información del copyright. Por ejemplo:

```
%%%----------------------------------------------------------------------
%%% Copyright Ericsson Telecom AB 1996
%%%
%%% All rights reserved. No part of this computer programs(s) may be
%%% used, reproduced,stored in any retrieval system, or transmitted,
%%% in any form or by any means, electronic, mechanical, photocopying,
%%% recording, or otherwise without prior written permission of
%%% Ericsson Telecom AB.
%%%----------------------------------------------------------------------
```

- **Cabeceras de archivos, revisión del historial:**

Cada archivo del código fuente debe ser documentado con cada revisión que se realice y en el que se enseñe quién cambió código del archivo y qué modificó. Por ejemplo:

```
%%%-------------------------------------------------------------------
%%% Revision History
%%%-------------------------------------------------------------------
%%% Rev PA1 Date 960230 Author Fred Bloggs (ETXXXXX)
%%% Intitial pre release. Functions for adding and deleting foobars
%%% are incomplete
%%%-------------------------------------------------------------------
%%% Rev A Date 960230 Author Johanna Johansson (ETXYYY)
%%% Added functions for adding and deleting foobars and changed
%%% data structures of foobars to allow for the needs of the Baz
%%% signalling system
%%%-------------------------------------------------------------------
```

- **Cabecera de archivo, descripción:**

Cada archivo debe empezar con una corta descripción del módulo que contenido y una breve descripción de todas las funciones exportadas. Por ejemplo:

```
%%%-------------------------------------------------------------------
%%% Description module foobar_data_manipulation
%%%-------------------------------------------------------------------
%%% Foobars are the basic elements in the Baz signalling system. The
%%% functions below are for manipulating that data of foobars and for
%%% etc etc etc
%%%-------------------------------------------------------------------
%%% Exports
%%%-------------------------------------------------------------------
%%% create_foobar(Parent, Type)
%%%   returns a new foobar object
%%%   etc etc etc
%%%-------------------------------------------------------------------
```

- **Usar un sistema controlador de código fuente:**

Todos los proyectos compuestos por un código fuente complicado necesitan un sistema de control como, **RCS, CVS** o **Clearcase** para mantener un seguimiento de todos los módulos.

CAPÍTULO 3

PLAN DE PROYECTO

Capítulo 3 – Plan de proyecto

En este capítulo se detalla el análisis realizado durante la fase inicial del proyecto, mostrando la planificación tanto temporal, económica, como de recursos de este. Como fase inicial de un proyecto de desarrollo de software, es necesario realizar un análisis de los principales requisitos, basados en los objetivos del proyecto, y que definen el sistema que se llevará a cabo.

Seguidamente, se lleva a cabo una planificación inicial de las tareas a realizar, una planificación temporal, económica y un análisis de las herramientas que se utilizarán y justificar finalmente los presupuestos.

Por último, para comenzar con las estimaciones se decide o especifica un método para el desarrollo del proyecto. En este caso se utiliza el método de desarrollo ágil de software.

3.1 – Captura de requisitos:

En este apartado, se realiza la captura de requisitos que serán necesarios para las estimaciones en los siguientes apartados de este capítulo.

3.1.1 – Requisitos funcionales:

Un requisito funcional, es un conjunto de entradas, comportamientos y/o salidas del sistema de software o de sus componentes. A continuación se detallan los de nuestro proyecto:

- Registrar el jugador en la partida con un nombre de identificación.
- Asignar un cartón de números a cada jugador registrado para jugar.
- Mostrar un log que comunique toda la información de la partida al jugador.
- Mostrar los números aleatorios que conforman la partida.
- Reclamar línea por parte del jugador.
- Reclamar bingo por parte del jugador.
- Salir de la partida y borrar jugador.

3.1.2 – Requisitos no funcionales:

Los requisitos no funcionales, a diferencia de los funcionales son requisitos que especifican criterios para juzgar las operaciones de un sistema, en lugar de sus comportamientos específicos:

Genéricos:

- Seguridad: Código programado para proteger el sistema de ataques de denegación de permiso y de intrusiones no deseadas.
- Disponibilidad: El sistema estará en funcionamiento 24 horas los 7 días de la semana.
- Portabilidad: El sistema funcionará en cualquier ordenador que cuente con Google Chrome.
- Usabilidad: Tendrá una interfaz intuitiva y simple.
- Escalabilidad: Posibilidad de añadir nuevas funcionalidades de forma sencilla y permitir que siga ampliándose.
- Estabilidad: Muy alta gracias a la autogestión de procesos.
- Concurrencia: Alta para poder afrontar partidas de miles de jugadores.
- Utilidad: Las opciones ofrecidas al jugador han de ser útiles.
- Manuales de usuario: Para que los jugadores conozcan cómo funciona el juego.

Sistema:

- Diseño: Además de las funcionalidades para el jugador, la interfaz es sencilla e intuitiva para facilitar su uso por parte de los usuarios.
- Fluidez: Gracias a la velocidad de la máquina virtual Erlang lanzando procesos.
- Margen de error: Muy pequeño, Erlang ofrece la posibilidad de crear sistemas muy robustos.
- Mantenibilidad: el sistema no exige un gran mantenimiento, si un proceso falla, volverá a lanzarse otro sin que se rompa el sistema. Los cambios y actualizaciones de ser necesarios podrán hacerse sin interrumpir el sistema.
- Rendimiento: Alto, gracias a la potencia de Erlang.
- Denegación de servicio: Google Chrome con soporte para WebSockets.

3.2 – Estimación temporal, diagrama de Gantt:

En la siguiente imagen podemos observar las fechas y los tiempos transcurridos en cada una de las fases de realización del proyecto. Las jornadas laborales han sido de 8 horas durante 5 días a la semana lo que nos queda en 40 horas semanales de trabajo.

	Name	Duration	Start	Finish	Predecessors
1	Aprendizaje y Estado del Art	21 days?	2/25/14 8:00 AM	3/25/14 5:00 PM	
2	Captura de Requisitos	5 days?	3/25/14 8:00 AM	3/31/14 5:00 PM	
3	**Planificación**	**2 days?**	**4/1/14 8:00 AM**	**4/2/14 5:00 PM**	**2**
4	Estimación	2 days	4/1/14 8:00 AM	4/2/14 5:00 PM	
5	Diagrama de Estimación	1 day?	4/1/14 8:00 AM	4/1/14 5:00 PM	
6	Herramientas	1 day?	4/1/14 8:00 AM	4/1/14 5:00 PM	
7	Presupuesto	1 day?	4/1/14 8:00 AM	4/1/14 5:00 PM	
8	Análisis	10 days	4/3/14 8:00 AM	4/16/14 5:00 PM	3
9	Diseño	15 days	4/17/14 8:00 AM	5/7/14 5:00 PM	8
10	**Implementación**	**65 days**	**4/17/14 8:00 AM**	**7/16/14 5:00 PM**	**8**
11	Lado Cliente	5 days	4/17/14 8:00 AM	4/23/14 5:00 PM	
12	Lado Servidor	65 days	4/17/14 8:00 AM	7/16/14 5:00 PM	
13	Pruebas	15 days	7/17/14 8:00 AM	8/6/14 5:00 PM	10
14	Documentación	15 days	8/7/14 8:00 AM	8/27/14 5:00 PM	13

Imagen 41. Fases del proyecto

A continuación se ve una imagen del diagrama de Gantt. En este diagrama destacamos la desigualdad en el tiempo de documentación y preparación frente al periodo de implementación, esta desigualdad es tan grande porque el proyecto se construyó siguiendo la Metodología Ágil que se explicará al final de este capítulo.

Las bases de esta metodología son la interacción con el cliente, el valor del equipo de programación (en este caso sólo hay un desarrollador) frente a la calidad del entorno para trabajar, el primar más el desarrollo sobre la documentación del software y ser flexibles a posibles cambios a la hora de desarrollar el proyecto, para ello se utilizan iteraciones cortas.

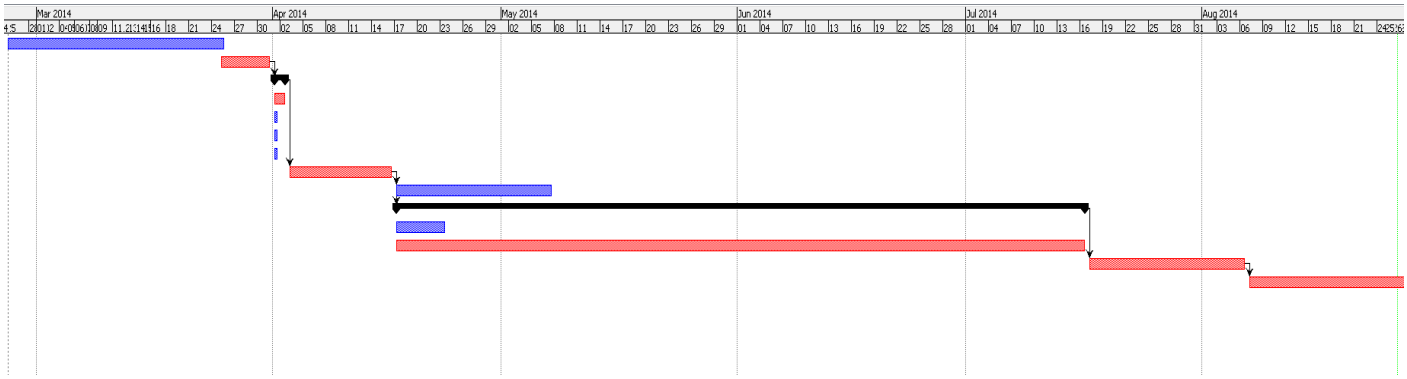


Imagen 42. Diagrama de Gantt

3.3 – Herramientas:

Una vez realizadas las estimaciones, procedemos a la elección de las herramientas software que van a ser requeridas para la implementación del sistema.

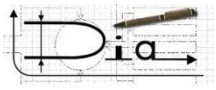
A continuación realizamos una breve descripción de los componentes software utilizados para la consecución del proyecto:

Ubuntu 12.04 LTS: Sistema Operativo que da soporte para instalar y ejecutar: Erlang, Gedit, Dia y Chrome.

Erlang: Lenguaje en el que se diseña la tecnología del lado del servidor.

Gedit: Aplicación libre que se usa para escribir el código de nuestro sistema.

Chrome: Navegador Web creado por Google, que utilizamos para la ejecución de nuestra sistema.



Dia: Aplicación para el desarrollo de diagramas personalizados.

Windows 7: Sistema Operativo que da soporte para instalar y ejecutar: Office 2010, OpenProj.

Microsoft Office 2010: Suite de oficina de Microsoft del que se usa el procesador de texto Microsoft Word para la redacción, editado y maquetación de la documentación del proyecto.

OpenProj : Software de administración de proyectos, para las estimaciones temporales entre otras, que posee una licencia libre.

Github: Sistema de control de versiones utilizado para alojar y progresar con la consecución del proyecto de forma segura.

3.4 – Presupuesto:

Dividiremos el presupuesto en tres partes, la primera describe el coste hardware, la segunda el coste del software utilizado y por último el coste del desarrollo (la mano de obra):

- Un ordenador portátil.
- Conexión a Internet durante el tiempo de trabajo.

Teniendo en cuenta que el tiempo del proyecto es de 6 meses según el diagrama de Gantt, obtenemos los siguientes costes:

HARDWARE	COSTE MENSUAL	COSTE TOTAL (€)
Ordenador personal	50 € (mes)	300 €
Conexión a internet	19,95 € (mes)	119,70 €
	TOTAL:	419,7 €

Tabla 1. Tabla de presupuesto hardware

En segundo lugar, calculamos los costes generados por el software utilizado para el desarrollo de la aplicación:

SOFTWARE	COSTE ANUAL	COSTE TOTAL (€)
Windows 7	300 € (año)	150 €
Ubuntu 12.04 LTS	-	-
Gedit	-	-
Microsoft Office 2007	129 € (año)	64,5 €
Github	-	-
Dia	-	-
OpenProj	-	-
	TOTAL:	214,5 €

Tabla 2. Tabla de presupuesto software

Por último, realizamos el coste del desarrollador del proyecto, teniendo en cuenta las siguientes condiciones:

- Consta de un desarrollador.
- Tomando la jornada laboral de 8 horas.

	TIEMPO	COSTE
Desarrollador	960 horas	12 € la hora
	TOTAL:	11520 €

Tabla 3. Tabla de presupuesto mano de obra

Calculados estos datos, realizamos la suma de todos los costes para obtener el precio final de la aplicación:

	COSTE
Hardware	419,7 €
Software	214,5 €
Desarrollo	11520 €
TOTAL	12154,2 €

Tabla 4. Tabla de presupuesto total

3.5 – Metodología:

El proyecto se ha construido siguiendo la metodología ágil en el desarrollo de software. El desarrollo de software no es una tarea fácil. Prueba de ello es que existen numerosas propuestas metodológicas que inciden en distintas dimensiones del proceso de desarrollo. Por una parte están aquellas propuestas más tradicionales que se centran especialmente en el control del proceso, estableciendo rigurosamente las actividades involucradas, los artefactos que se deben producir, y las herramientas y notaciones que se usarán. Estas propuestas han demostrado ser efectivas y necesarias en un gran número de proyectos, pero también han presentado problemas en otros muchos. Una posible mejora es incluir en los procesos de desarrollo más actividades, más artefactos y más restricciones, basándose en los puntos débiles detectados. Sin embargo, el resultado final sería un proceso de desarrollo más complejo que puede incluso limitar la propia habilidad del equipo para llevar a cabo el proyecto. Otra aproximación es centrarse en otras dimensiones, como por ejemplo el factor humano o el producto software. Esta es la filosofía de las metodologías ágiles, las cuales dan mayor valor al individuo, a la colaboración con el cliente y al desarrollo incremental del software con iteraciones muy cortas.

Las metodologías ágiles están revolucionando la manera de producir software.

En 2001 tras una reunión celebrada en EEUU, nace el término "ágil" aplicado al desarrollo de software. En esta reunión participaban un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores e impulsores de metodologías de software. En esta reunión se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

Tras esta reunión se creó *The Agile Alliance*[5], una organización, sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones para que adopten dichos conceptos. El punto de partida es el Manifiesto Ágil, que resume la filosofía de esta metodología.

[5] Organización sin ánimo de lucro con el fin de avanzar en el desarrollo de metodologías ágiles (http://www.agilealliance.org/).

3.5.1 – El Manifiesto Ágil.

Según el Manifiesto se valora:

- **Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas**. La gente es el principal factor de éxito de un proyecto software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente. Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.

- **Desarrollar software que funciona más que conseguir una buena documentación.** La regla a seguir es "no producir documentos a menos que sean necesarios de forma inmediata para tomar un decisión importante". Estos documentos deben ser cortos y centrarse en lo fundamental.

- **La colaboración con el cliente más que la negociación de un contrato.** Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.

- **Responder a los cambios más que seguir estrictamente un plan.** La habilidad de responder a los cambios que puedan surgir a los largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

Los valores anteriores inspiran los doce principios del manifiesto. Son características que diferencian un proceso ágil de uno tradicional. Los dos primeros principios son generales y resumen gran parte del espíritu ágil. El resto tienen que ver con el proceso a seguir y con el equipo de desarrollo, en cuanto metas a seguir y organización del mismo. Los principios son:

I. La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.

II. Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.

III. Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.

IV. La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.

V. Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.

VI. El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.

VII. El software que funciona es la medida principal de progreso.

VIII. Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.

IX. La atención continua a la calidad técnica y al buen diseño mejora la agilidad.

X. La simplicidad es esencial.

XI. Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.

XII. En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo, y según esto ajusta su comportamiento.

3.5.2 – Comparación:

La siguiente tabla recoge esquemáticamente las principales diferencias de las metodologías ágiles con respecto a las tradicionales. Estas diferencias que afectan no sólo al proceso en sí, sino también al contexto del equipo así como a su organización.

Metodologías Ágiles	Metodologías Tradicionales
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambios
Impuestas internamente (por el equipo)	Impuestas externamente
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas/normas
No existe contrato tradicional o al menos es bastante flexible	Existe un contrato prefijado
El cliente es parte del equipo de desarrollo	El cliente interactúa con el equipo de desarrollo mediante reuniones
Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio	Grupos grandes y posiblemente distribuidos
Pocos artefactos	Más artefactos
Pocos roles	Más roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

Tabla 5. Metodologías Ágiles vs Metodologías tradicionales

CAPÍTULO 4

ANÁLISIS

En este cuarto capítulo, analizaremos los objetivos y requisitos del sistema, a continuación mostraremos sus tablas para finalizar mostrando los distintos diagramas de casos de uso y los diagramas de secuencia del sistema.

4.1 – Objetivos y Requisitos:

4.1.1 – Especificación de objetivos:

Comenzamos con la especificación de los objetivos:

OBJ-01	Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento.
Versión	*1.0*
Autor	*Manuel Rico López*
Descripción	*El sistema deberá soportar y gestionar el juego de miles de jugadores online al mismo tiempo*
Subobjetivos	*SubOBJ-01 < Registrarse en el juego >* *SubOBJ-02 < Gestionar la reclamación de línea >* *SubOBJ-03 < Gestionar la reclamación de bingo >*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 6. OBJ-01

SubOBJ-01	Registrarse en el juego.
Versión	*1.0*
Autor	*Manuel Rico López*
Descripción	*El sistema deberá ser capaz de otorgar un nombre de identificación a un jugador.*
Estado	*Completado.*
Comentarios	*Sistema de registro del juego.*

Tabla 7. SubOBJ-01

SubOBJ-02	Gestionar la reclamación de línea.
Versión	*1.0*
Autor	*Manuel Rico López*
Descripción	*El sistema deberá ser capaz de detectar y validar la reclamación de una línea por parte del jugador.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 8. SubOBJ-02

SubOBJ-03	Gestionar la reclamación de bingo.
Versión	*1.0*
Autor	*Manuel Rico López*
Descripción	*El sistema deberá ser capaz de detectar y validar la reclamación de un bingo por parte del jugador.*
Estado	*Completado.*
Comentarios	*Listados generales, en detalle y operaciones de compra.*

Tabla 9. SubOBJ-03

4.1.2 – Especificación de Requisitos:

En las siguientes tablas veremos los requisitos funcionales:

FR-01	Sistema de registro del usuario.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento>*
Requisitos asociados	*FR-02 <Registrarse en la aplicación > NFR-01 <Seguridad> NFR-02 <Disponibilidad> NFR-04 <Usabilidad> NFR-08 <Utilidad> NFR-09 <Manuales de usuario> NFR-10 <Diseño> NFR-11 <Fluidez> NFR-12 <Mantenibilidad> IRQ-01 <Usuarios>*
Descripción	*El sistema deberá registrar con un nombre al jugador.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 10. FR-01

FR-02	Asignar cartón de números al jugador registrado para jugar.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento>*
Requisitos asociados	*FR-02 <Registrarse en la aplicación > NFR-01 <Seguridad> NFR-02 <Disponibilidad> NFR-04 <Usabilidad> NFR-08 <Utilidad> NFR-09 <Manuales de usuario> NFR-10 <Diseño> NFR-11 <Fluidez> NFR-12 <Mantenibilidad> IRQ-01 <Usuarios>*
Descripción	*El sistema deberá repartir al jugador un cartón de números para que pueda jugar.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 11. FR-02

FR-03	Mostrar un log que comunique toda la información de la partida al jugador.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento>*
Requisitos asociados	*FR-02 <Registrarse en la aplicación > NFR-01 <Seguridad> NFR-02 <Disponibilidad> NFR-04 <Usabilidad> NFR-07 <Concurrencia> NFR-08 <Utilidad> NFR-09 <Manuales de usuario> NFR-10 <Diseño> NFR-11 <Fluidez> NFR-12 <Mantenibilidad> NFR-13 <Rendimiento> IRQ-01 <Usuarios>*
Descripción	*El sistema deberá ofrecer toda la información de la partida al jugador.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 12. FR-03

FR-04	Mostrar los números aleatorios que conforman la partida.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento>*
Requisitos asociados	*FR-02 <Registrarse en la aplicación > NFR-01 <Seguridad> NFR-02 <Disponibilidad> NFR-04 <Usabilidad> NFR-08 <Utilidad> NFR-09 <Manuales de usuario> NFR-10 <Diseño> NFR-11 <Fluidez> NFR-12 <Mantenibilidad> IRQ-01 <Usuarios>*
Descripción	*El sistema deberá mostrar los números aleatorios que conforman la partida de bingo.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 13. FR-04

FR-05	Reclamar línea por parte del jugador.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento>*
Requisitos asociados	*FR-02 <Registrarse en la aplicación > NFR-01 <Seguridad> NFR-02 <Disponibilidad> NFR-04 <Usabilidad> NFR-07 <Concurrencia> NFR-08 <Utilidad> NFR-09 <Manuales de usuario> NFR-10 <Diseño> NFR-11 <Fluidez> NFR-12 <Mantenibilidad> NFR-13 <Rendimiento> IRQ-01 <Usuarios>*
Descripción	*El sistema deberá recoger y validar la petición de línea del jugador.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 14. FR-05

FR-06	Reclamar bingo por parte del jugador.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento>*
Requisitos asociados	*FR-02 <Registrarse en la aplicación > NFR-01 <Seguridad> NFR-02 <Disponibilidad> NFR-04 <Usabilidad> NFR-07 <Concurrencia> NFR-08 <Utilidad> NFR-09 <Manuales de usuario> NFR-10 <Diseño> NFR-11 <Fluidez> NFR-12 <Mantenibilidad> NFR-13 <Rendimiento> IRQ-01 <Usuarios>*
Descripción	*El sistema deberá recoger y validar la petición de bingo del jugador.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 15. FR-06

FR-07	Salir de la partida y borrar jugador.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento>*
Requisitos asociados	*FR-02 <Registrarse en la aplicación > NFR-01 <Seguridad> NFR-02 <Disponibilidad> NFR-04 <Usabilidad> NFR-08 <Utilidad> NFR-09 <Manuales de usuario> NFR-10 <Diseño> NFR-11 <Fluidez> IRQ-01 <Usuarios>*
Descripción	*El sistema deberá permitir salir de la partida a los jugadores, cuando un jugador sale de la partida, su registro se borrará inmediatamente de la tabla ETS.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 16.FR-07

El siguiente paso es especificar también los requisitos no funcionales:

NFR-01	Seguridad.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema entenderá que una persona no registrada es un **observador** y no podrá interactuar en la partida a no ser que se registre. A través del uso de cláusulas el sistema se protege de ataques de denegación de servicio y otras actividades no deseadas.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 17. NFR-01

NFR-02	Disponibilidad.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá gestionar la aplicación de forma que se encuentre disponible las 24 horas y los 7 días de la semana.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 18. NFR-02

NFR-03	Portabilidad.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	El sistema funcionará en cualquier ordenador que cuente con Google Chrome.
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 19. NFR-03

NFR-04	Usabilidad.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá poseer una interfaz de usuario intuitiva y simple.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 20. NFR-04

NFR-05	Escalabilidad.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento>*
Descripción	*El sistema deberá poseer la facilidad de añadir nuevas funcionalidades y permitir que siga ampliándose.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 21. NFR-05

NFR-06	Estabilidad.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá ser estable para no interrumpir las partidas. Esto se consigue gracias a la tecnología Erlang y su gestión de procesos.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 22. NFR-06

NFR-07	Concurrencia.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá ser altamente concurrente para soportar partidas de miles de jugadores.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 23. NFR-07

NFR-08	Utilidad.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá solamente poseer funcionalidades que sean útiles para su uso y para el usuario.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 24. NFR-08

NFR-09	Manuales de Usuario.
.Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá poseer interfaces intuitivas, un diseño sencillo, de fácil uso y modificación.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 25. NFR-09

NFR-10	Diseño
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá poseer interfaces intuitivas, un diseño, sencillo, de fácil uso, y modificación.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 26. NFR-10

NFR-11	Fluidez
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*Este requisito se llevará a cabo mediante el paso de mensajes en Erlang.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 27. NFR-11

NFR-12	Margen de Error.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá contar con un margen de error muy pequeño, para no arrastrar fallos de funcionamiento.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 28. NFR-12

NFR-13	Mantenibilidad.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá ser fácil de mantener. En cuanto a fallos, si un proceso falla, volverá a lanzarse otro sin que se rompa el sistema. Tanto los cambios como las actualizaciones podrán hacerse sin interrumpir el funcionamiento del sistema.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 29. NFR-13

NFR-14	Rendimiento.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá de ofrecer un alto rendimiento para dar servicio a miles de jugadores al mismo tiempo.*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 30. NFR-14

Para terminar con los requisitos, se muestran las tablas de los requisitos de información:

IRQ - 01	Usuario.
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá almacenar información sobre los jugadores.*
Datos específicos	*- Nombre de jugador* *- Número de identificación de proceso (Pid)* *- Cartón asignado*
Estado	*Completado.*
Comentarios	*Ninguno.*

Tabla 31. IRQ-01

4.2 – Diagramas de casos de uso:

Siguiendo con el análisis, se muestran ahora los diagramas de casos de uso desarrollados a partir de los requisitos anteriores. A continuación se muestra el diagrama de casos de uso de una partida de bingo:

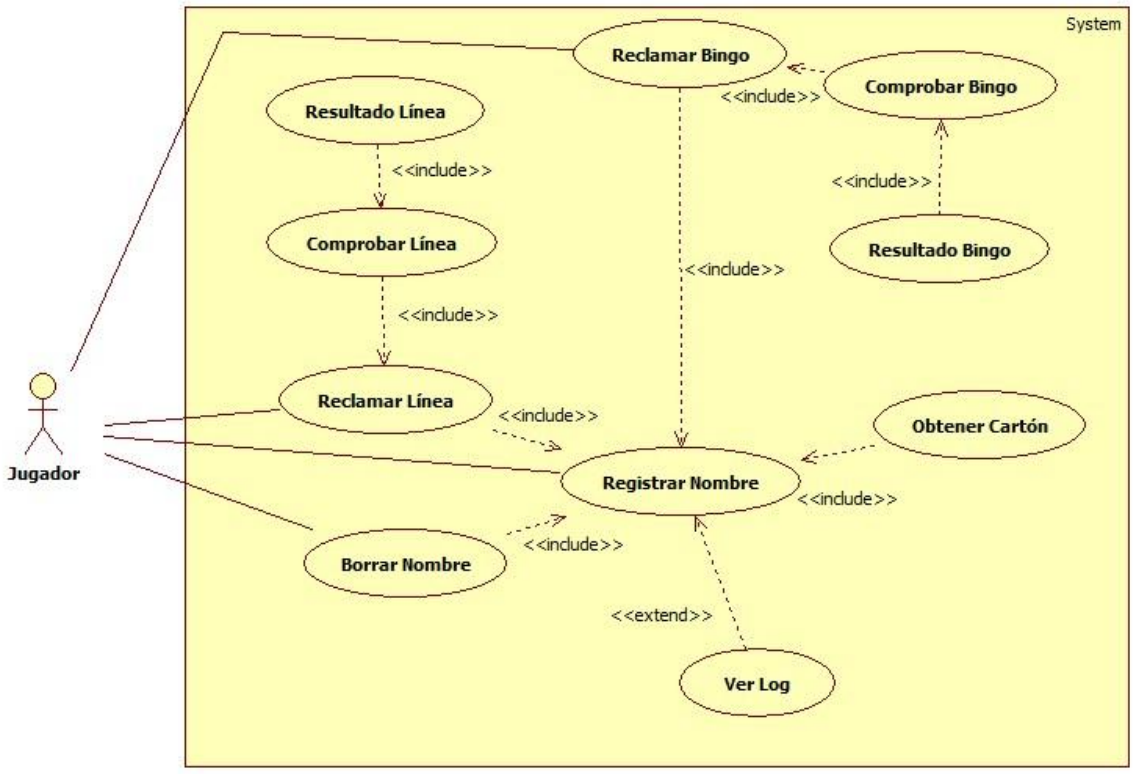


Imagen 43. Diagrama de casos de uso

El usuario, si quiere llegar a ser un jugador y dejar de ser un observador, deberá registrarse con un nombre (“Registrar Nombre”) para así obtener un cartón (“Obtener Cartón”) y una identificación para jugar.

Una vez que la partida empiece, el jugador puede pulsar los botones Bingo (“Reclamar Bingo”) y Línea (“Reclamar Línea”) según crea conveniente para su partida, realizándose sendas validaciones (“Comprobar Línea” y “Comprobar Bingo”) y recibiendo cada validación su correspondiente resultado (“Resultado Línea” y “Resultado Bingo”).

Tras jugar varias partidas, el jugador abandonará el juego cerrando el navegador o la pestaña en la que se está llevando a cabo la partida, lo que provocará el inmediato borrado del jugador (“Borrar Nombre”) de la tabla ETS que explicaremos más adelante.

El observador, como su nombre indica, no puede hacer más que observar el cómo transcurre la partida a través de la información que aporta el log (“Ver Log”).

4.2.1 – Actores:

ACT - 01	Jugador
Versión	*1.0*
Autor	*Manuel Rico López*
Descripción	*Este actor representa al jugador que accede a la web para comenzar una partida y se registra con un nombre de jugador.*
Comentarios	*Ninguno.*

Tabla 32. Actor jugador

ACT - 02	Observador
Versión	*1.0*
Autor	*Manuel Rico López*
Descripción	*Este actor representa al jugador que accede a la web para comenzar una partida y no se registra con ningún nombre, para dejar de ser observador debe registrarse con un nombre y esperar a que comience una nueva partida.*
Comentarios	*Ninguno.*

Tabla 33. Actor observador

4.2.2 – Especificaciones de casos de uso:

De los casos de uso anteriores, vamos a realizar su análisis, mediante la especificación de estos a través de tablas:

UC - 01	Registrar Nombre
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá comportarse tal y como se describe en el siguiente caso de uso cuando un usuario registra un nombre de jugador.*
Precondición	*El usuario debe entrar en la web, en el estado de observador*
Secuencia normal	**Paso – Acción** *1 – El usuario accede a la web del bingo.* *2 – Es necesario que el usuario se registre con un nombre de jugador para jugar.* *3 – El usuario introducirá su nombre de jugador y pulsará "Apuntarse".* *4 – El sistema registrará al usuario en la tabla ETS y le facilitará un cartón .*
Postcondición	*Debe mantenerse en todo momento registrado con su nombre.*
Excepciones	**Paso – Acción** *1 – Tras acceder a la web el usuario, si no se registra con un nombre, será interpretado por el sistema como observador.*
Comentarios	*Ninguno.*

Tabla 34. Caso de uso registrar nombre

UC - 02	Reclamar Línea
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá comportarse tal y como se describe en el siguiente caso de uso cuando un usuario reclama una línea.*
Precondición	*El usuario debe entrar en la web y registrarse con un nombre de jugador.*
Secuencia normal	**Paso – Acción** *1 – El usuario accede a la web del bingo.* *2 – Es necesario que el usuario se registre con un nombre de jugador para jugar.* *3 – El usuario introducirá su nombre de jugador y pulsará "Apuntarse".* *4 – El sistema registrará al usuario en la tabla ETS y le facilitará un cartón.* *5 – El sistema recibirá la reclamación de línea tras ser pulsado el botón "Linea".* *6 – El sistema comprobará si la línea es correcta o no, devolviendo el resultado mostrándolo en el log.*
Postcondición	*Debe mantenerse en todo momento registrado con su nombre.*
Excepciones	**Paso – Acción** *2 – El jugador entra en la web y no se registra.*
Comentarios	*Ninguno.*

Tabla 35. Caso de uso registrar línea

UC - 03	Reclamar Bingo
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá comportarse tal y como se describe en el siguiente caso de uso cuando un usuario reclama un bingo.*
Precondición	*El usuario debe entrar en la web y registrarse con un nombre de jugador.*
Secuencia normal	**Paso – Acción** *1 – El usuario accede a la web del bingo.* *2 – Es necesario que el usuario se registre con un nombre de jugador para jugar.* *3 – El usuario introducirá su nombre de jugador y pulsará "Apuntarse".* *4 – El sistema registrará al usuario en la tabla ETS y le facilitará un cartón.* *5 – El sistema recibirá la reclamación de bingo tras ser pulsado el botón "Bingo".* *6 – El sistema comprobará si el bingo es correcto o no, devolviendo el resultado mostrándolo en el log.*
Postcondición	*Debe mantenerse en todo momento registrado con su nombre.*
Excepciones	**Paso – Acción** *2 – El jugador entra en la web y no se registra.*
Comentarios	*Ninguno.*

Tabla 36. Caso de uso reclamar bingo

UC - 04	Borrar Nombre
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá comportarse tal y como se describe en el siguiente caso de uso cuando un usuario borra su nombre abandonando la partida.*
Precondición	*El usuario debe entrar en la web y registrarse con un nombre de jugador.*
Secuencia normal	**Paso – Acción** *1 – El usuario accede a la web del bingo.* *2 – Es necesario que el usuario se registre con un nombre de jugador para jugar.* *3 – El usuario abandonará la partida cerrando el navegador o la pestaña en la que se esté jugando la partida, el sistema borrará automáticamente el registro de ese jugador de la tabla ETS.*
Postcondición	*Ninguna.*
Excepciones	**Paso – Acción** *2 – El jugador entra en la web y no se registra.*
Comentarios	*Ninguno.*

Tabla 37. Caso de uso borrar nombre

UC - 05	Ver Log
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá comportarse tal y como se describe en el siguiente caso de uso.*
Precondición	*El usuario debe entrar en la web.*
Secuencia normal	**Paso – Acción** *1 – El usuario accede a la web del bingo.* *2 –.El jugador no registrado (observador) verá toda la información de la partida en el log. El jugador registrado con un nombre, verá toda la información de sus acciones y de las acciones de los demás jugadores.*
Postcondición	*Ninguna.*
Excepciones	**Paso – Acción** *Ninguna.*
Comentarios	*Ninguno.*

Tabla 38. Caso de uso ver log

UC - 06	Obtener Cartón
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá comportarse tal y como se describe en el siguiente caso de uso cuando un usuario se registra con nombre de jugador.*
Precondición	*El usuario debe entrar en la web y registrarse con nombre de jugador.*
Secuencia normal	**Paso – Acción** *1 – El usuario accede a la web del bingo.* *2 – El jugador se registra con nombre de jugador.* *3 – El usuario introducirá su nombre de jugador y pulsará "Apuntarse".* *4 – El sistema registrará al usuario en la tabla ETS y le facilitará un cartón.*
Postcondición	*El usuario debe mantenerse registrado con un nombre de jugador*
Excepciones	**Paso – Acción** *2 – El jugador entra en la web y no se registra.*
Comentarios	*Ninguno.*

Tabla 39. Caso de uso obtener carton

UC - 07	Comprobar Línea
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá comportarse tal y como se describe en el siguiente caso de uso cuando el sistema recibe una reclamación de línea.*
Precondición	*El usuario debe entrar en la web y registrarse con nombre de jugador.*
Secuencia normal	**Paso – Acción** *1 – El usuario accede a la web del bingo.* *2 – El jugador se registra con nombre de jugador.* *3 – El usuario introducirá su nombre de jugador y pulsará "Apuntarse".* *4 – El sistema registrará al usuario en la tabla ETS y le facilitará un cartón.* *5 – El jugador pulsará el botón "Linea".* *6 – El sistema comprobará si la línea es válida o no y le devolverá el resultado mostrado por el log.*
Postcondición	*El usuario debe mantenerse registrado con un nombre de jugador.*
Excepciones	**Paso – Acción** *2 – El jugador entra en la web y no se registra.*
Comentarios	*Ninguno.*

Tabla 40. Caso de uso comprobar línea

UC - 08	Comprobar Bingo
Versión	*1.0*
Autor	*Manuel Rico López*
Objetivos asociados	*OBJ-01 < Obtención de un sistema online capaz de modelar un juego de bingo de alto rendimiento >*
Descripción	*El sistema deberá comportarse tal y como se describe en el siguiente caso de uso cuando el sistema recibe una reclamación de bingo.*
Precondición	*El usuario debe entrar en la web y registrarse con nombre de jugador.*
Secuencia normal	**Paso – Acción** *1 – El usuario accede a la web del bingo.* *2 – El jugador se registra con nombre de jugador.* *3 – El usuario introducirá su nombre de jugador y pulsará "Apuntarse".* *4 – El sistema registrará al usuario en la tabla ETS y le facilitará un cartón.* *5 – El jugador pulsará el botón "Bingo".* *6 – El sistema comprobará si el bingo es válida o no y le devolverá el resultado mostrado por el log.*
Postcondición	*El usuario debe mantenerse registrado con un nombre de jugador.*
Excepciones	**Paso – Acción** *2 – El jugador entra en la web y no se registra.*
Comentarios	*Ninguno.*

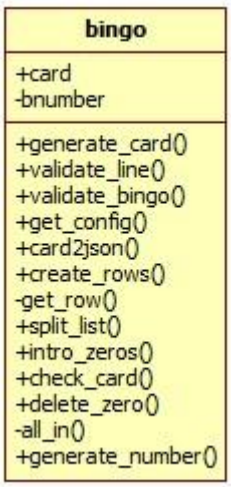
Tabla 41. Caso de uso comprobar bingo

4.3 – Módulos del sistema:

Como explicamos en los capítulos iniciales de la documentación, los sistemas Erlang deben desarrollarse utilizando módulos. Los cuales deben ser lo más pequeños y fáciles de leer posibles.

- **Módulo bingo:**

El módulo bingo es una librería con varios métodos que nos aportarán distintas funcionalidades a nuestro sistema.



En la imagen observamos el módulo bingo, compuesto por los tipos:

- "card" (cartón): de tipo diccionario. El tipo diccionario es un tipo abstracto de datos que provee de un mapeo clave-valor, en otros lenguajes el equivalente podrían ser las tablas hash o arrays asociativos.

  Se exporta para que pueda ser usado por otros módulos.

  La manera que se utilizó en el proyecto para crear un diccionario que diese lugar a nuestro cartón de juego es usando la función ***`dict:from_list/1([{clave, valor}])`*** para introducir un par clave-valor, en este caso, número de fila del cartón-serie de números de la fila.

- "bnumber": de tipo "pos_integer" (entero siempre positivo).

A continuación vemos con un + las funciones exportadas para que puedan ser invocadas desde otros módulos.

Las funciones que cabe destacar son las siguientes:

- "generate_card": usada para generar un cartón con números aleatorios.

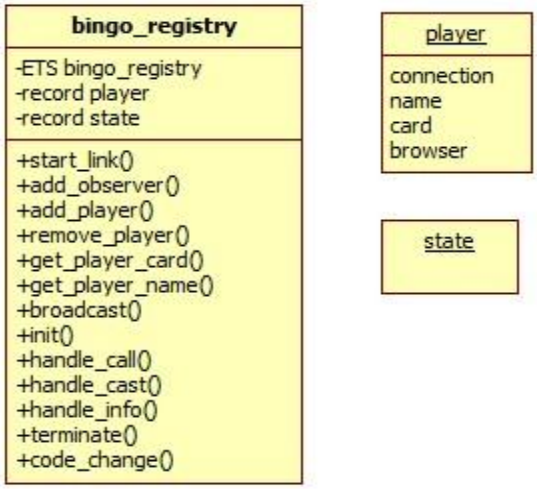
- "get_config": usada para configurar parámetros de la partida (número mínimo requerido de jugadores para empezar una partida, número de segundos en la cuenta atrás, tiempo entre números, etc…)

- "check_card": para comprobar validaciones tanto de línea como de bingo.

- "generate_number": para generar un número aleatorio para la partida.

- **Módulo bingo_registry:**

El módulo bingo_registry es el que se encarga de realizar las operaciones con la tabla ETS. El comportamiento de este módulo es el de un "gen_server" (como explicamos anteriormente en el Capítulo 2 de la documentación en el Apartado 2.8.1) con los registros player y state, funciones y tabla ETS que recibe el mismo nombre que el módulo (en este mismo capítulo, explicaremos a continuación qué son las tablas ETS).



De este grupo de funciones, cabe destacar:

- "start_link": donde se lanza el proceso gen_server ligado.

- "init": donde se crea la tabla ETS para guardar los registros de los jugadores.

- "add_player": usado para añadir un jugador a la partida una vez se haya registrado.

- "remove_player": usado para borrar el jugador de la tabla ETS una vez abandone el juego.

- "get_player_card": usado para obtener el cartón que se le ha asignado al jugador.

- "get_player_name": usado para obtener el nombre del jugador asociado a su pid.

- "broadcast": usado para enviar la información de la partida a observadores y jugadores.

En cuanto a los registros contamos con el state del gen_server vacío (ya que en este módulo no se cambia en ningún momento de estado ni se necesita almacenar información en él, sólo se trabaja con la tabla ETS) y con el registro player, este registro cuenta con los campos:

- "connection": que nos aporta el pid (identificador de proceso) del observador ó del jugador.

- "name": donde se guardará el nombre con el que el jugador fue registrado.

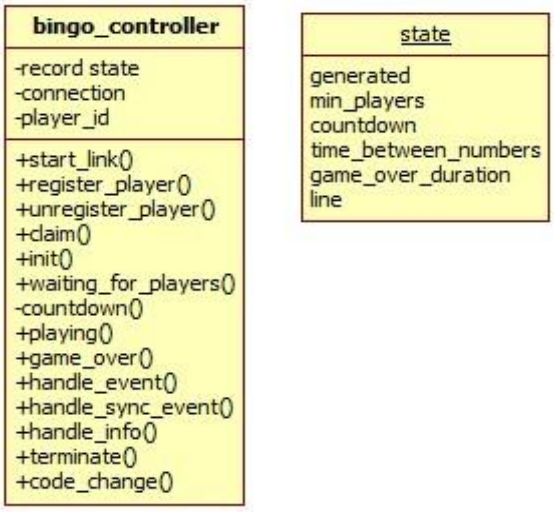
- "card": donde se almacena el cartón asignado al jugador después de registrarse para comenzar la partida.

- "browser": donde se guarda la información del navegador que está usando el jugador.

- **Módulo bingo_controller:**

El módulo bingo_controller es la máquina de estados finita (se ha explicado un ejemplo en el Capítulo 2 de la documentación, Apartado 2.8.2) que se encarga de controlar toda la lógica de la partida.

Este módulo está compuesto por un estado del gen_fsm, unos atributos tipo "connection" que es un pid y "player_id" que es una iolist.



El registro state está formado por los siguientes campos:

- "generated": es una lista donde se almacenan los números que han sido escogidos aleatoriamente durante el transcurso de la partida, se usa posteriormente para poder realizar las validaciones de línea y bingo.

- "min_players": en este campo indicamos el número mínimo de jugadores necesarios para iniciar la partida.

- "countdown": se indica los segundos que tardará en cerrarse el registro de jugadores e iniciarse la partida una vez que se registra el número mínimo de jugadores.

- "time_between_number": refleja el tiempo de espera entre los números generados aleatoriamente para jugar.

- "game_over_duration": especifica el tiempo de espera una vez se termina una partida.

- "line": bandera (booleano) utilizada para controlar las llamadas de línea de forma que sólo pueda cantarse una línea válida por partida.

En el bingo_controller podemos separar las funciones en varios grupos, las funciones del API y los callbacks del gen_fsm que se ocupan del arranque del servidor, del manejo de los eventos síncronos, de los eventos asíncronos, de los eventos sea cual sea el estado de la máquina y por último las funciones de parada.

- Funciones del API:
    - “start_link”: crea un proceso gen_fsm que llama al init para inicializarse.
    - “register_player”: registra un jugador en la partida.
    - “unregister_player”: borra el registro del jugador que abandona la partida.
    - “claim”: para reclamar una línea o un bingo.
- Callbacks de arranque:
    - “init”: sirve para arrancar el gen_fsm con los parámetros configurados.
- Callbacks asíncronos:
    - “countdown”: se encarga de realizar la cuenta atrás para que empiece la partida.
    - “waiting_for_players”: se encarga de continuar generando números aleatorios para la partida en caso de que varios jugadores abandonen el juego sin ser los suficientes como para que se pare la partida.
    - “playing”: controla la partida mientras se esté jugando.
    - “game_over: se encarga de reiniciar la partida y vuelve a poner la bandera “line” a valor falso para comenzar otra partida.
- Callbacks síncronos:
    - “countdown”: se encarga del registro del jugador y de enviar la notificación del registro mientras el estado de la máquina es countdown.
    - “waiting_for_players”: se encarga del registro del jugador y de enviar la notificación del registro mientras el estado de la máquina es waiting_for_players, también controla que el número de jugadores registrados sea el adecuado para comenzar la partida.
    - “playing”: se encarga de manejar las peticiones de línea y de bingo de los jugadores que conforman la partida.

- Las funciones de parada y actualización de código en caliente son:
    - "terminate": para parar el servidor.
    - "code_change": para actualizar el código sin necesidad de parar el sistema.

Si observamos los nombres de los callbacks, podemos darnos que cuenta de que muchos coinciden con los estados de la máquina y que por tanto, tienen el mismo nombre para los callbacks asíncronos como para los callbacks síncronos. Este problema Erlang lo soluciona haciendo "pattern matching" en las cláusulas, es decir, si el número y tipo de argumentos coincide con el countdown asíncrono, se ejecutará este y no el countdown síncrono.

A continuación mostraremos un diagrama de estados de nuestra máquina gen_fsm.

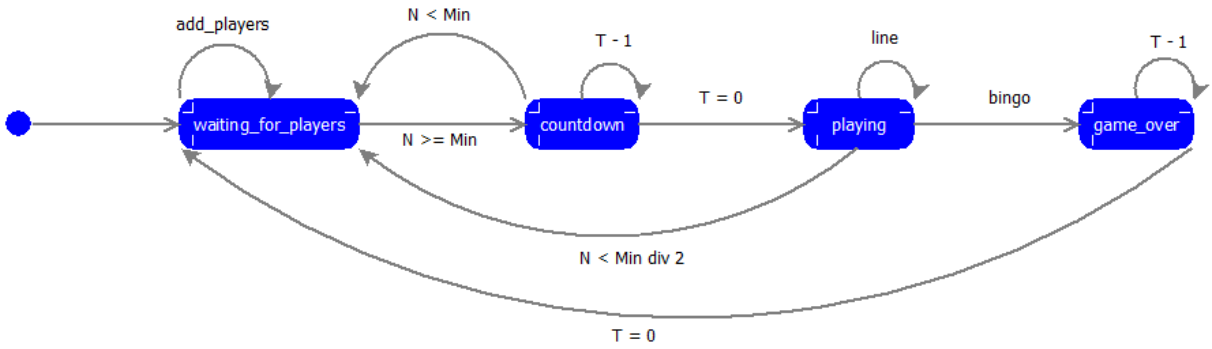


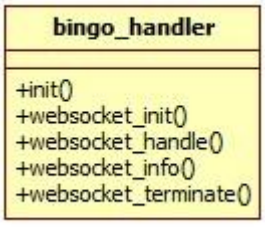
Imagen 44. Diagrama de estados bingo_controller

En este diagrama de estados podemos observar las transiciones y las condiciones de nuestra máquina de estados finita.

- **Módulo bingo_handler:**

Este módulo tiene un comportamiento "cowboy_websocket_handler" que se encarga de manejar las comunicaciones entre cliente y servidor mediante objetos JSON.

Las funciones que destacamos de este módulo son las que manejan el comportamiento del websocket:



- "websocket_init": recoge la información que interesa de las cabeceras y las compacta para optimizar el espacio de la información, es así como sabemos qué navegador utiliza el usuario y enviamos esta información y el pid al bingo_registry.

- "websocket_handle": maneja la información tramitada desde el cliente por el usuario, el registrarse con un nombre o reclamar una línea o un bingo.

- "websocket_info": maneja la información enviada por el sistema de forma asíncrona sin que participe en nada el usuario, como por ejemplo los números generados aleatoriamente para la partida.

- "websocket_terminate": es el encargado de manejar la eliminación de un registro de jugador cuando este abandona el juego.

4.4 - Diagramas de secuencia (se mostrarán en inglés por coherencia con las buenas prácticas)**:**

En los diagramas de secuencia se destaca la ordenación temporal de los mensajes enviados y recibidos por el sistema y los usuarios (o administradores). Son construidos a partir de los objetos que participan en la interacción, mostrando los mensajes en orden que realizan.

- **Diagrama de secuencia: registrar un jugador.**

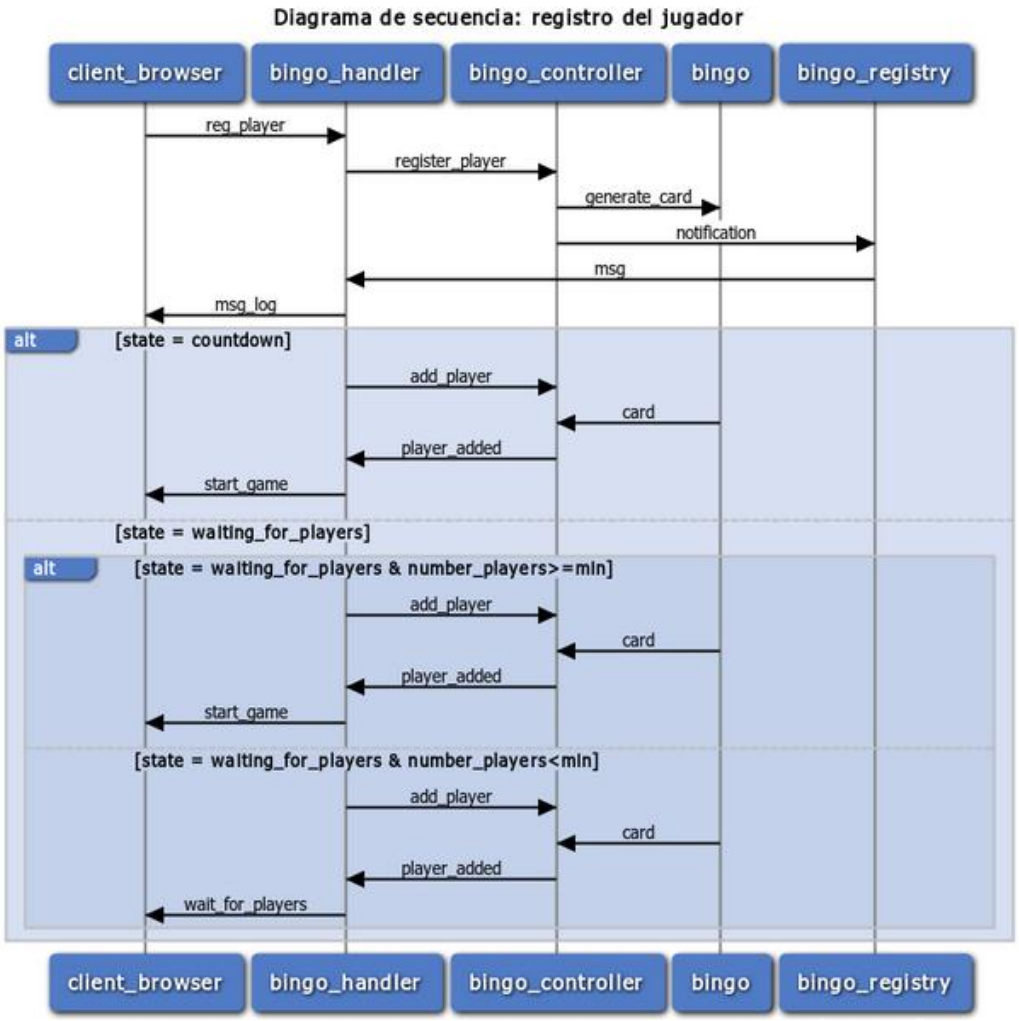


Imagen 45. Diagrama de secuencia de registro de jugador

En este primer diagrama observamos el intercambio de mensajes entre el lado cliente y el servidor cuando un usuario registra su nombre para poder comenzar una partida.

El comportamiento del registro del jugador depende en qué estado de la máquina de estados finita (“bingo_controller”) se encuentre.

Si el estado de la máquina es “countdown” se añadirá al jugador con su respectivo cartón a la partida. Para esto, el jugador desde el navegador comunica al manejador (“bingo_handler”), que es el encargado de manejar las comunicaciones entre cliente y servidor, que quiere registrarse con un nombre de jugador, el manejador se lo comunica al controlador (“bingo_controller”) que se encarga de la lógica del transcurso de la partida. Desde el controlador, se le pide a la librería bingo que genere un cartón para el jugador registrado y a continuación se genera un mensaje de notificación (“notification”) que el “bingo_registry”, que se encarga de manejar la tabla ETS con los registros de los jugadores, mandará al manejador para que este lo reenvíe al cliente y así pueda obtener el mensaje en el log de que se ha unido a la partida y pueda comenzar a jugar esta con su cartón y su nombre de jugador.

Es importante destacar, que el bingo_registry, al asociar el pid del proceso que representa la conexión con el cartón que se obtiene en el módulo bingo, posibilita al sistema realizar la comprobación y validación del cartón del lado del servidor, por lo que no puede ser forzada la comprobación del lado cliente mediante código malicioso.

En el caso de que la partida no haya comenzado y esté en estado “waiting_for_players” a la espera de jugadores, la máquina de estados estará en este estado mientras no se lleguen a registrar un mínimo y configurable número de jugadores.

Si la máquina se encuentra en este estado y se han registrado el mínimo de jugadores en la partida, el jugador se registrará comunicándoselo así al manejador, el manejador se lo transmite al controlador que a su vez requerirá a la librería bingo que genere un cartón para ese jugador, esta librería se lo envía y de este modo, como el número de jugadores es igual o mayor que el número mínimo configurado para comenzar la partida, la partida empieza.

Si la máquina de estados se encuentra en espera de jugadores pero no se han registrado los superiores, el registro de jugadores seguirá el mismo proceso que en el caso anterior, con la diferencia de que el jugador registrado esperará con su cartón asignado a que se registren más jugadores hasta cumplir con la condición del número mínimo para comenzar la partida.

- **Diagrama de secuencia: validar línea:**

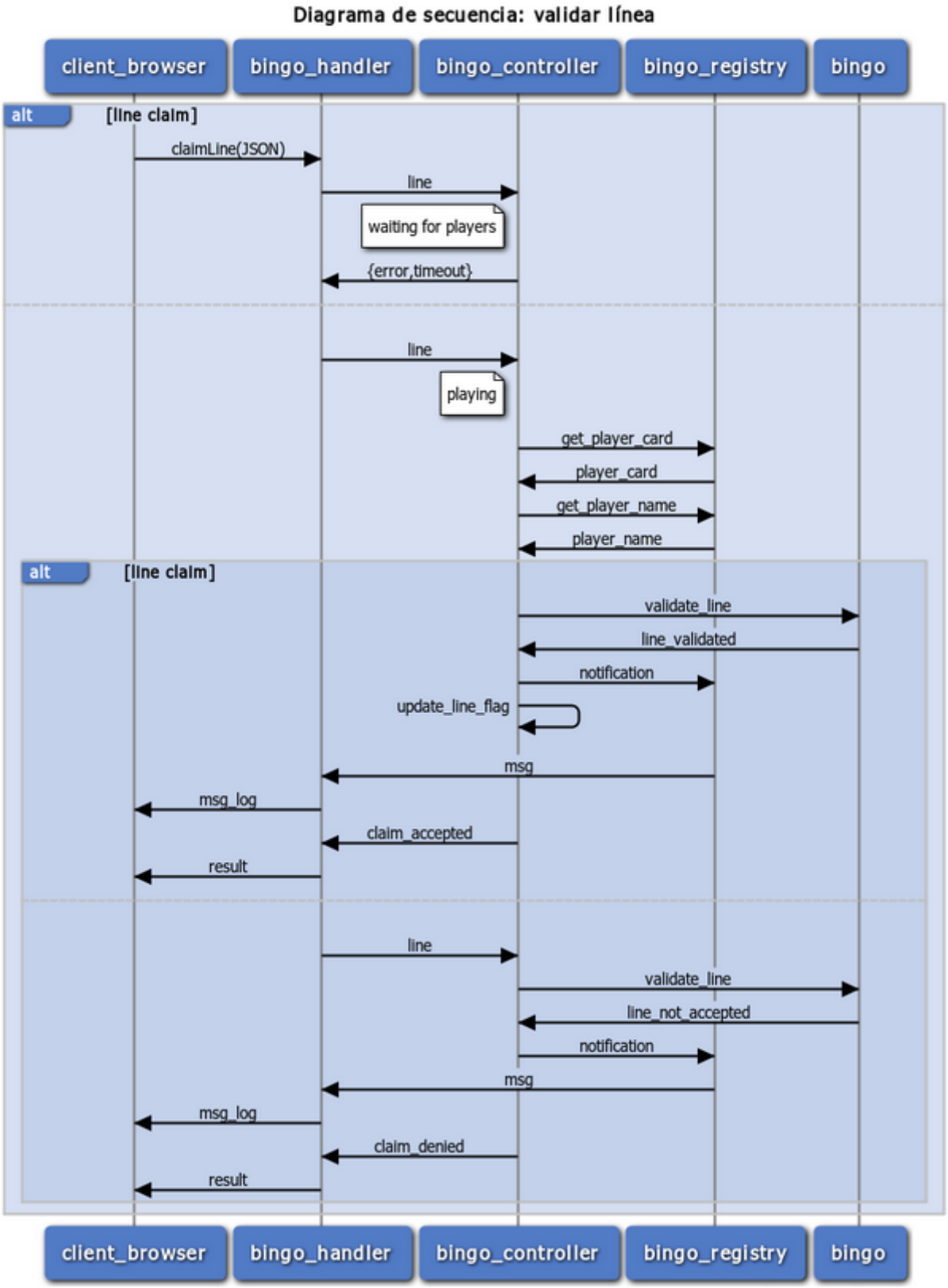


Imagen 46. Diagrama de secuencia de validar una línea

En este diagrama observamos el comportamiento del sistema a la hora de reclamar una línea por parte del jugador.

En el primer caso, el cliente enviará al manejador un mensaje “claimLine”, el manejador lo recibe y se comunica con el controlador para aplicar la lógica de la partida, el controlador comprueba en qué estado se encuentra la máquina de estados, si se encuentra en “waiting for players” devuelve un error con la tupla {error, timeout} que el sistema ignora (esto se hace por seguridad, de modo que si un usuario un grupo de ellos pulsan repetidas veces un botón de petición de línea, el sistema los ignora protegiéndose así mismo de un bloqueo por denegación de servicio). Si la máquina de estado se encuentra en estado “playing”, transcurriendo así la partida, el controlador solicitará al “bingo_registry” la tarjeta y el nombre del jugador para realizar la posterior validación, de la validación se encarga la librería “bingo”, si la validación se realiza y la línea cantada es correcta, el controlador se lo comunica a “bingo_registry” y cambia la bandera “flag” del atributo line del estado de la máquina, impidiendo así que se puedan cantar más líneas válidas a posteriori. “bingo_registry” se encargará de enviar el mensaje al manejador, que le transmitirá el resultado al cliente a través del log.

En el caso de que la línea no pase el proceso de validación, el atributo “line” del estado del controlador no cambiará, permitiendo que se cante línea a posteriori, siguiendo los mismos pasos de transmisión de mensaje, se comunica al jugador que su línea no ha sido válida a través del log.

Es importante explicar que en el módulo bingo_controller, sólo actúa al mismo tiempo un proceso bingo_controller por lo que el paso de los mensajes de los jugadores se procesarán secuencialmente por este proceso bingo_controller. Esta es la razón por la que aunque 100 jugadores presionen línea al mismo tiempo, sólo uno puede ganar.

- **Diagrama de secuencia: validar bingo:**

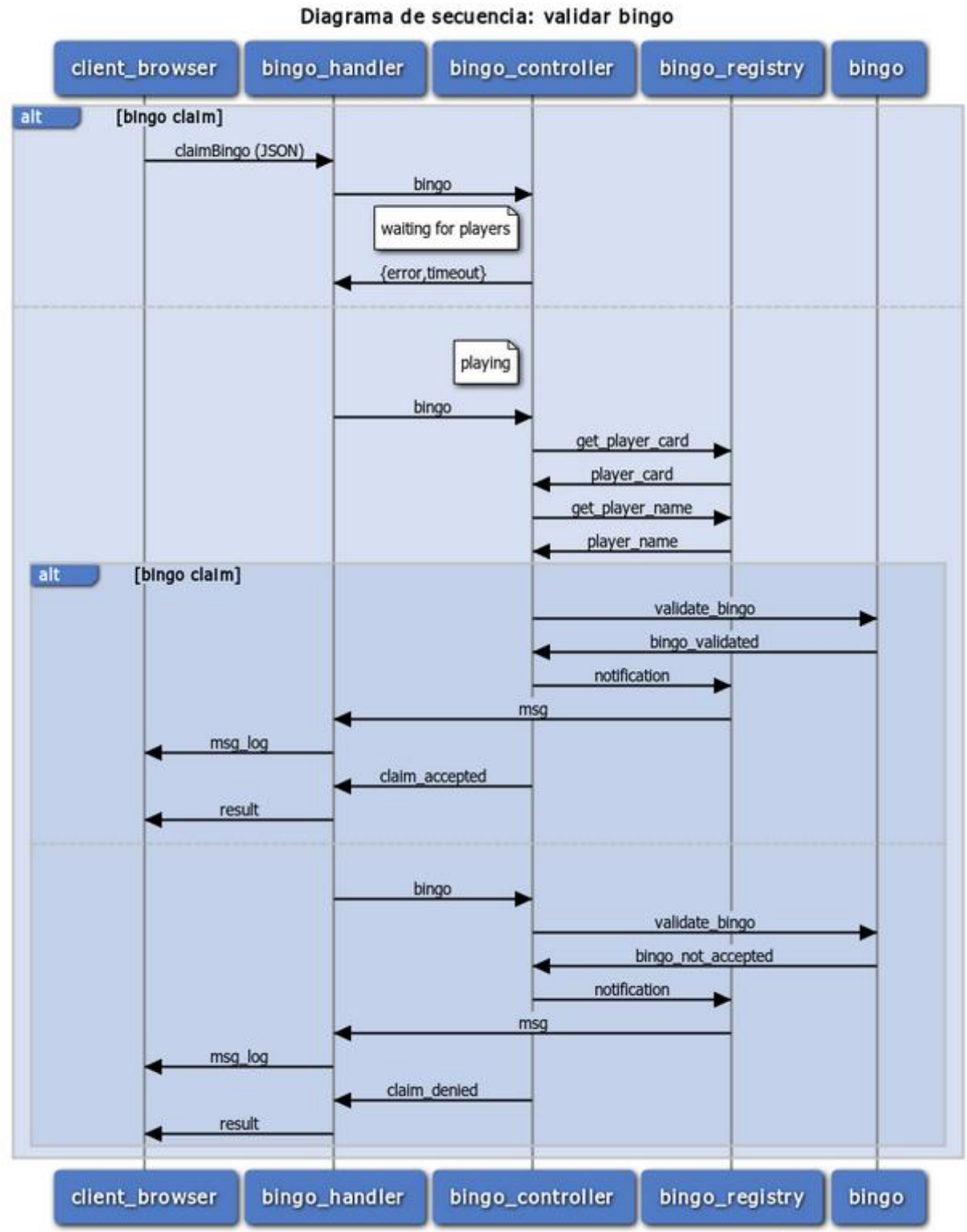


Imagen 47. Diagrama de secuencia de validar un bingo

En este diagrama observamos el comportamiento del sistema a la hora de reclamar un bingo por parte del jugador.

En el primer caso, el cliente enviará al manejador un mensaje “claimBingo”, el manejador lo recibe y se comunica con el controlador para aplicar la lógica de la partida, el controlador comprueba en qué estado se encuentra la máquina de estados, si se encuentra en “waiting for players” devuelve un error con la tupla {error, timeout} que el sistema ignora (esto se hace por seguridad, de modo que si un usuario un grupo de ellos pulsan repetidas veces un botón de petición de bingo, el sistema los ignora protegiéndose así mismo de un bloqueo por denegación de servicio). Si la máquina de estado se encuentra en estado “playing”, transcurriendo así la partida, el controlador solicitará al “bingo_registry” la tarjeta y el nombre del jugador para realizar la posterior validación, de la validación se encarga la librería “bingo”, si la validación se realiza y el bingo cantado es correcto, el controlador se lo comunica a “bingo_registry” que se encargará de enviar el mensaje al manejador, que le transmitirá el resultado al cliente a través del log finalizando así la partida.

En el caso de que el bingo no pase el proceso de validación, se siguen los mismos pasos de transmisión de mensaje que se siguieron con la validación correcta pero se comunica al jugador que su bingo no ha sido válido a través del log.

- **Diagrama de secuencia: borrar jugador.**

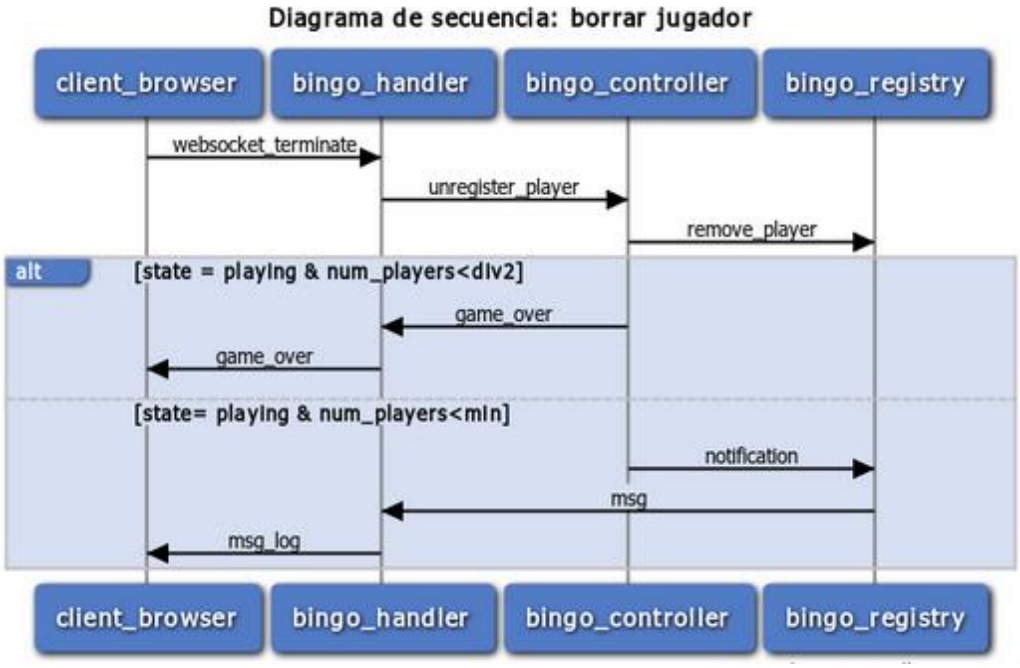


Imagen 48. Diagrama de secuencia de borrado de jugador

En este diagrama podemos ver cómo sale un jugador de la partida borrando así su nombre de la tabla ETS.

Al cerrar la pestaña del navegador donde se lleva a cabo la partida, o al cerrar completamente el navegador, el cliente manda una señal “websocket_terminate” al manejador, que se lo comunica al controlador para que dé de baja el registro de ese jugador,

este invoca el método “remove_player” de “bingo_registry” el cual borrará el registro de dicho jugador.

Si un jugador se borra de la partida y el número total restante de jugadores se menor que la división entera entre 2 del mínimo necesario de jugadores para empezar una partida, la partida se detiene y la máquina pasa al estado “game_over”.

Si un jugador abandona la partida borrando así su registro pero el número restante de jugadores es menor al mínimo configurable

CAPÍTULO 5

DISEÑO

Capítulo 5 - Diseño

5.1 – Arquitectura:

En este capítulo hablaremos sobre las arquitecturas lógica y física de nuestro sistema online de forma detallada donde podremos obtener un pleno conocimiento sobre la arquitectura completa del proyecto.

5.1.1 – Arquitectura lógica:

En este apartado explicaremos la arquitectura lógica del proyecto.

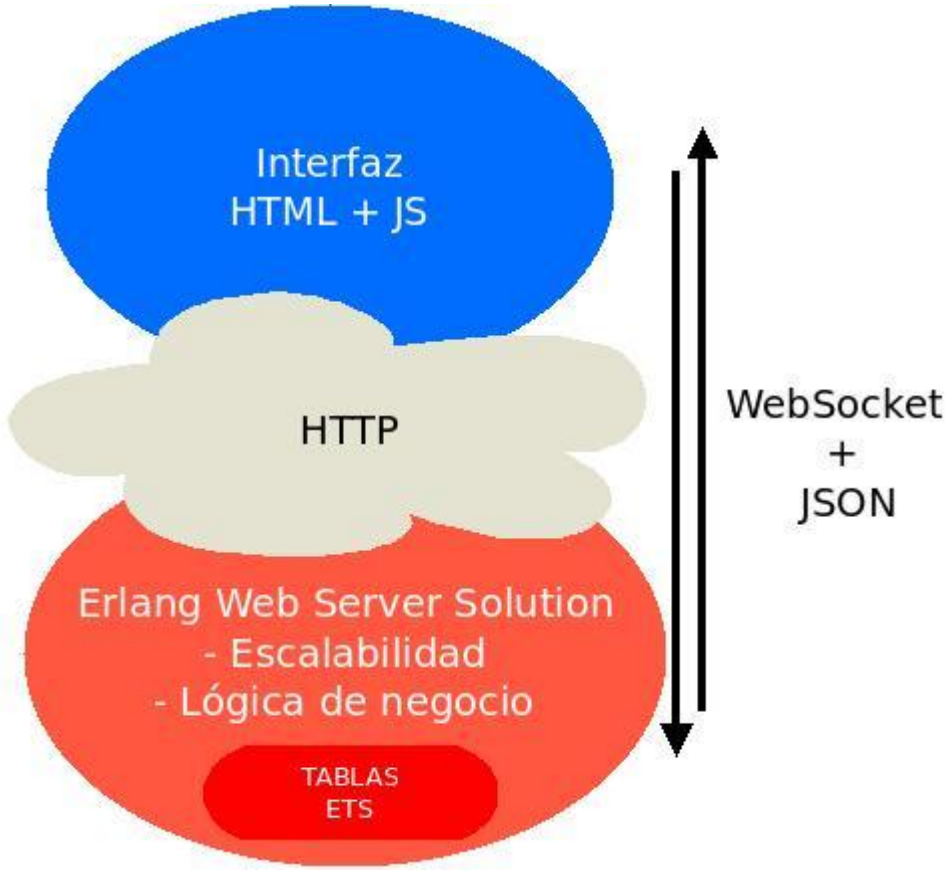


Imagen 49. Arquitectura lógica

En esta imagen observamos como las tecnologías que componen el lado cliente son el lenguaje de etiquetas HTML y JAVASCRIPT que se comunican con el lado servidor compuesto con el lenguaje Erlang mediante el uso del protocolo Websocket a través de peticiones HTTP usando objetos JSON (JavaScript Object Notation). A continuación procedemos a explicar más a fondo qué es un websocket y qué es JSON.

- **¿Qué es un websocket?**

Websockets es la tecnología que llega para resolver los problemas de comunicación que plantea el esquema anteriormente descrito. De manera simple, Websockets permite comunicar el cliente y el servidor a través de un canal Full Duplex bidireccional y sin tener que hacer polling por parte del cliente.

Veamos el esquema de Websockets.

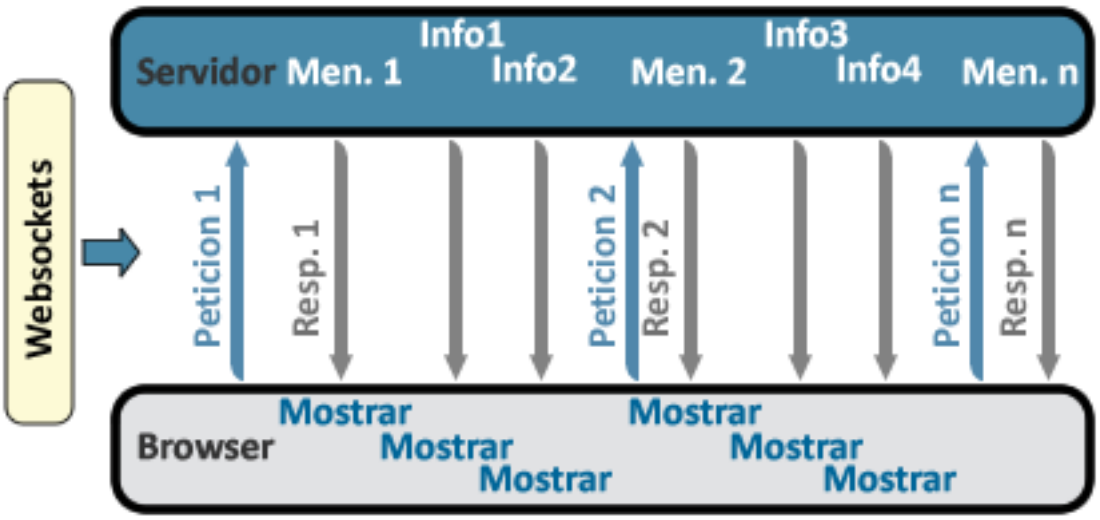


Imagen 50. Funcionamiento de un Websocket

Aquí podemos ver que esta tecnología nos permite intercambiar información entre el cliente y el servidor cuando cualquiera de las dos partes así lo requiera, sin necesidad de que el cliente esté sondeando al servidor, y con la ventaja extra de que las partes pueden enviar información al mismo tiempo por el mismo canal ya que al ser una conexión Full Duplex los mensajes no "chocarán".

Pero esta tecnología ofrece otras ventajas. Websockets es un estándar de HTML5, la API de WebSocket está siendo normalizada por el **W3C** (World Wide Web Consortium), y el protocolo WebSocket, a su vez, está siendo normalizado por el **IETF** (Internet Engineering Task Force) lo que hace que esta tecnología no tenga problemas de compatibilidad.

Ya no es necesario mezclar tecnologías extrañas para lograr asincronismo y nuestro código será más fácil de crear y mantener, eso sin mencionar que la lógica de aplicar Websockets es más sencilla de entender por tratarse de una mayor abstracción. La tecnología de Websockets se extiende por la mayoría de los servidores web más conocidos como Apache, Jetty, Tomcat, Tornado, Node.js, etc.

En el siguiente gráfico vemos una de las grandes ventajas de Websockets sobre la técnica de polling.

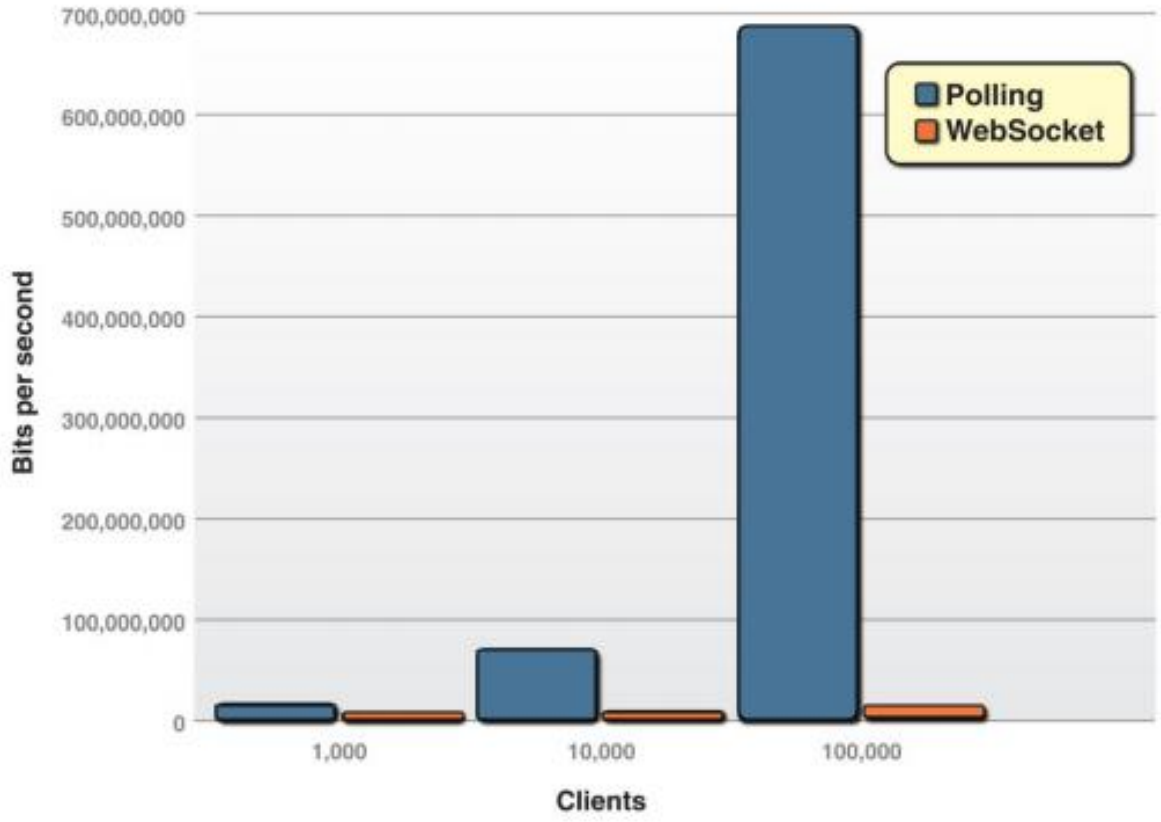


Imagen 51. Websocket Vs Polling

Aquí puede verse la reducción de tráfico innecesario en la red gracias a que no hay que implementar polling enviando paquetes de datos que solo "preguntan" al servidor si hay información para que este envíe al cliente. Eso sin contar los paquetes enviados por el cliente que terminan siendo inútiles porque el servidor no tiene información para enviar.

- **¿Cómo se implementa un Websocket?**

La implementación de Websockets si bien varía dependiendo del lenguaje de programación utilizado en el cliente o en el servidor la estructura básica es muy similar. Se trata de crear un socket del lado cliente y utilizarlo para enviar datos al servidor utilizando el lenguaje de interfaz que queramos, sea JSON, XML u otros.

Del lado del servidor se implementan las funciones o eventos que manejarán la información enviada a través del websocket.

Ejemplo en JavaScript.

Lado Cliente

```
   var websocket = new WebSocket("URL");

$(document).on("ready", iniciar);

function iniciar(){
  websocket.on("mensajeDesdeServidor", recibirMensaje);
       $('#botonEnviarCliente').click(enviarCliente);
}

function enviarCliente(e){

  var cliente = {
    dni: $("#dniCliente").val(),
    nombre: $("#nombreCliente").val(),
    apellido: $("#apellidoCliente").val()
  };
  var clienteEnviar = JSON.stringify(cliente);

    e.preventDefault();
  websocket.send("insertarCliente", clienteEnviar);
}

function recibirMensaje(datosServidor){
  $("#formulario label").text(datosServidor);
}
```

Lado Servidor

El lado del servidor es muy variado, si bien el ejemplo del lado cliente está dado con Javascript porque es el lenguaje más utilizado, el lado servidor implementa muchos y variados lenguajes, pero básicamente consta en recibir los paquetes que llegan con nombre de los eventos y el servidor sabe qué hacer en cada caso.
Un ejemplo en Javascript para Node.js del lado servidor.

```
   servidor.sockets.on("connection", arranque);

function arranque(usuario){
  usuario.on("insertarCliente", insertarClient);
}
   function insertarClient(claseJson){
  var clienteJson = claseJson;
  var clienteTemp = JSON.parse(clienteJson);
```

```
  client.query(
   'INSERT INTO cliente SET dni = ?, nombre = ?, apellido =
?',
   [clienteTemp.dni, clienteTemp.nombre, clienteTemp.apellido]
  );
}
```

Para explicar un poco el ejemplo vamos a situarnos en el código del lado cliente. Se crea una nueva variable de la clase WebSocket y se le entrega como parámetro la URL del servidor al que queremos conectarnos y opcionalmente el puerto y el protocolo (dependiendo de la librería utilizada), luego ese websocket representa nuestra conexión con el servidor. En la función iniciar se implementa que cuando el socket reciba un mensaje con encabezado "mensajeDesdeServidor" se lanza la función "recibirMensaje" que imprime por pantalla los datos devueltos por el servidor en la variable datosServidor.

Por otro lado indicamos que cuando se dispare el evento "click" del botón "botonEnviarCliente" se dispare la función "enviarCliente" que codifica un string JSON y lo envía al servidor con el encabezado "insertarCliente".

Cabe aclarar que no es necesario utilizar JSON o XML para transportar los datos, también puede hacerse de manera plana como strings.

Del lado Servidor se indica que cuando este inicie se dispare la función "arranque" y en esta se indica que cuando llegue un mensaje por el socket con el encabezado "inserterCliente" se dispare la función "insertarClient" que recibe como parámetro el objeto JSON enviado por el cliente.

Como se puede ver la lógica es sencilla, solo se establece la comunicación y luego se indican los eventos a disparar en cada caso.

Con Websockets como estándar la comunicación cliente servidor se realiza de una mejor manera, minimizando el tráfico innecesario en la red y permitiendo que ambas partes envíen datos cuando lo deseen lo que permite una nueva generación de aplicaciones web sumamente interactivas, de tiempo real y muy escalables.

- **¿Qué es JSON?**

JSON (Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos. Es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidas por los programadores de un extenso grupo de lenguajes como la familia C (C, C++, C#), Java, JavaScript, Perl, Python, Erlang, etc… Estas propiedades son las que hacen que JSON sea un lenguaje ideal para el intercambio de datos.

JSON está constituido por dos estructuras:

- Una colección de pares de nombre/valor. En varios lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un array asociativo.

- Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arrays, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

En JSON se presentan las siguientes formas:

- Un objeto es un conjunto desordenado de pares nombre/valor. Un objeto comienza con "{" y termine con "}". Cada nombre es seguido por ":" y los pares nombre/valor están separados por ",".

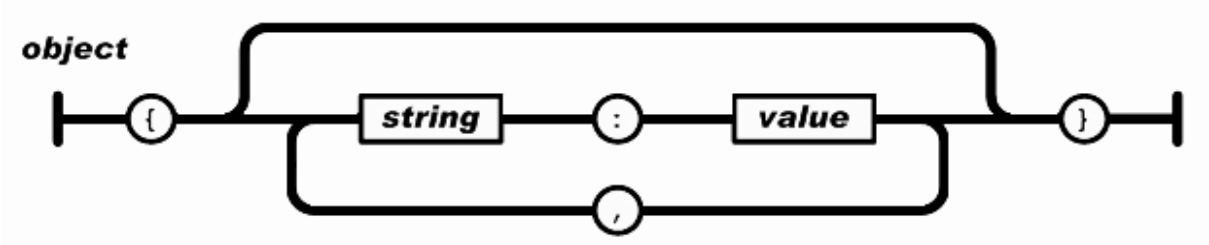


Imagen 52. Esquema Objeto JSON

- Un arreglo es una colección de valores. Un arreglo comienza con [(corchete izquierdo) y termina con] (corchete derecho). Los valores se separan por , (coma).

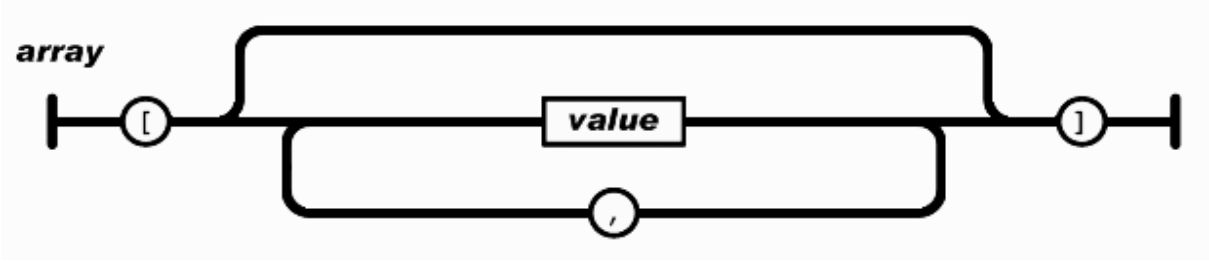


Imagen 53. Esquema Array JSON

- Un valor puede ser una cadena de caracteres con comillas dobles, o un número, o true o false o null, o un objeto o un arreglo. Estas estructuras pueden anidarse.

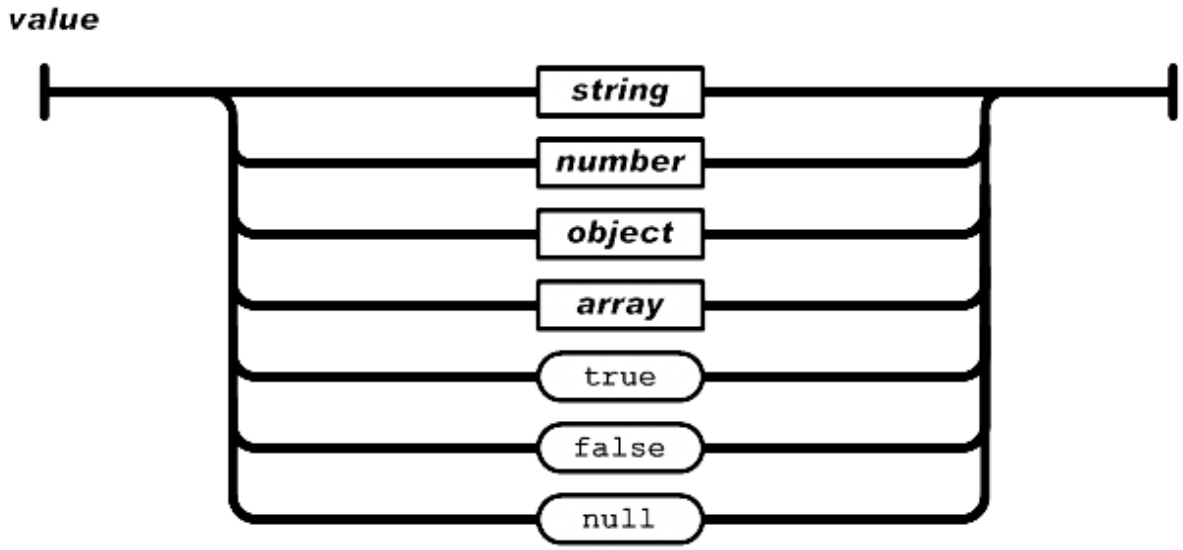


Imagen 54. Esquema valor JSON

- Una cadena de caracteres es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape. Un carácter está representado por una cadena de caracteres de un único carácter. Una cadena de carateres es parecida a una cadena de caracteres C o Java.

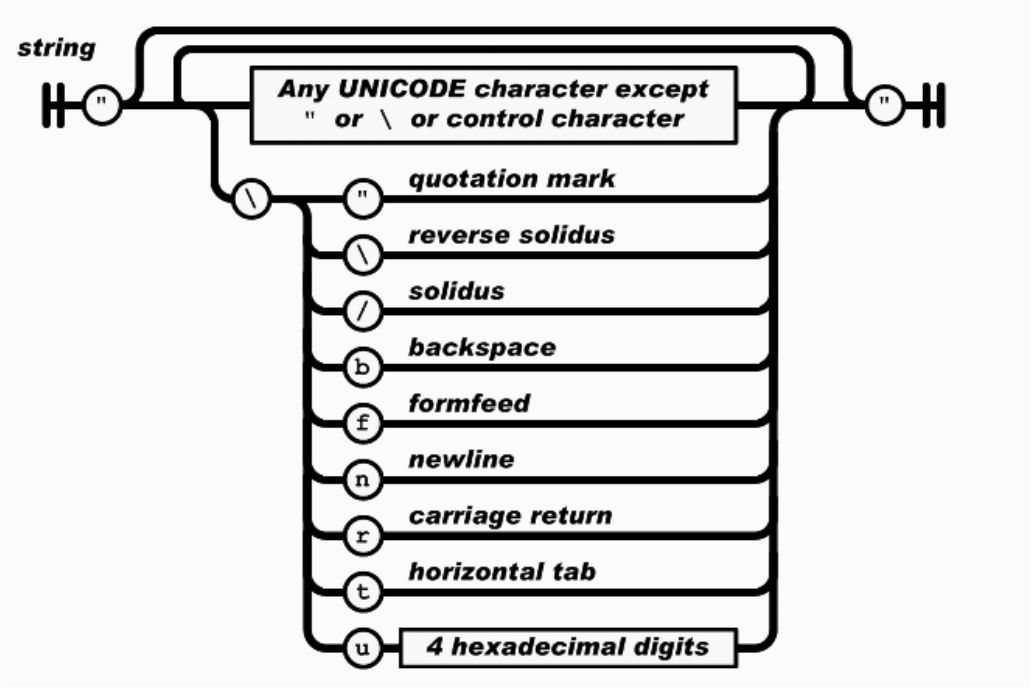


Imagen 55. Esquema cadena JSON

- Un número es similar a un número C o Java, excepto que no se usan los formatos octales y hexadecimales.

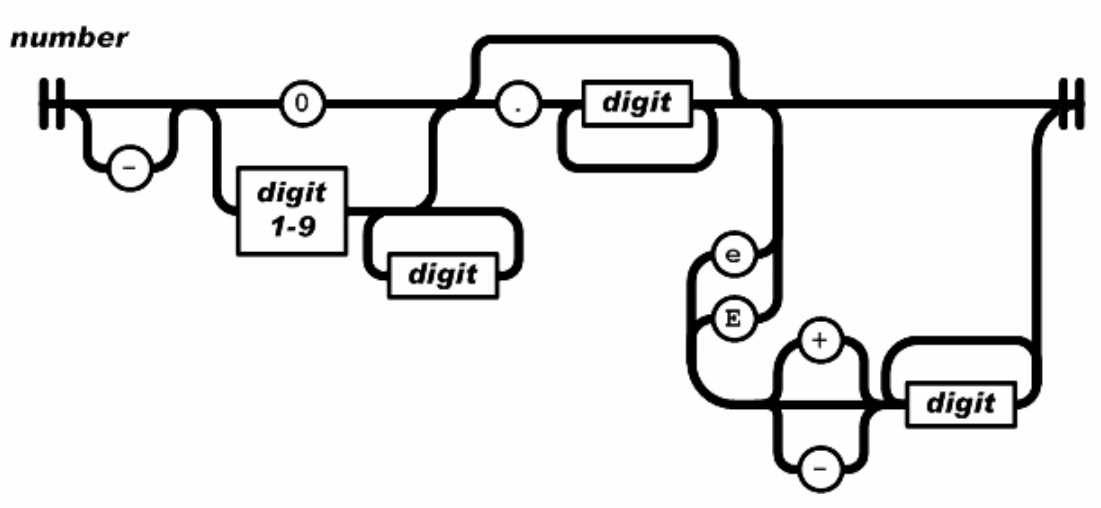


Imagen 56. Esquema número JSON

Los espacios en blanco pueden insertarse entre cualquier par de símbolos.

- **Métodos usados con JSON:**

Para aplicar un método sobre un objeto JSON debemos escribir `JSON.[method]`

Dos de los métodos más importantes y utilizados en JSON son el método **stringify** y el método **parse.**

- **Método stringify:** Convierte un valor de JavaScript en una cadena de notación de objetos JavaScript (JSON).

  ```
  JSON.stringify(value [, replacer] [, space])
  ```

  En esta línea de código le pasamos al método stringify los siguientes parámetros:

  - "value" : es obligatorio. Es un valor de JavaScript, normalmente un objeto o una matriz.

  - "replacer" : es opcional. Es una función o una matriz que transforma los resultados.

    Si *replacer* es una función, *JSON.stringify* llama a la función, pasando la clave y el valor de cada miembro. El valor devuelto se utiliza en lugar del valor original. Si la función devuelve *undefinied,* excluyen el miembro. La clave del objeto raíz es una cadena vacía *""*.

    Si *replacer* es una matriz, únicamente los miembros con valores de clave de la matriz se convertirán. El orden en que los miembros se

convierten en el mismo que el de las claves de la matriz. Se omite la matriz de *replacer* cuando el argumento de *value* también es una matriz.

- "space" : es opcional. Agrega la sangría, el espacio en blanco y caracteres de salto de línea al texto del valor devuelto JSON para que sea más fácil de leer.

  Si se omite *space*, el texto del valor devuelto se genera sin ningún tipo de espacio en blanco adicional.

  Si *space* es un número, al texto del valor devuelto se le aplica sangría al número especificado de espacios en blanco en cada nivel. Si *space* es mayor que 10, al texto se le aplica una sangría de 10 espacios.

  Si *space* es una cadena no vacía, como "\t", el texto del valor devuelto se aplica sangría con los caracteres de la cadena en cada nivel.

  Si *space* es una cadena mayor a 10 caracteres, se utilizan los primeros 10 caracteres.

El valor devuelto por JSON.stringify es una cadena que contiene el texto de JSON.

- **Método parse:** Convierte una cadena de la notación de objetos de JavaScript (JSON) en un objeto.

```
JSON.parse(text [, reviver])
```

Los parámetros de este método son:

- "text" : es obligatorio. Es una cadena de JSON válida.

- "reviver" : es opcional. Es la función que transforma los resultados. Se llama a esta función para cada miembro del objeto. Si un miembro contiene objetos anidados, estos se transforman antes que el objeto principal. Para cada miembro, ocurre lo siguiente:

  Si *reviver* devuelve un valor válido, el valor del miembro se reemplaza con el valor transformado, si devuelve el mismo valor que recibió, el valor del miembro no se modifica y por último, si *reviver* devuelve **null o undefinied**, el miembro se elimina.

El valor devuelto por JSON.parse es un objeto o una matriz.

5.1.2 – Arquitectura física:

En este apartado del capítulo, se explicarán los componentes físicos (hardware) necesarios y recomendados para el buen funcionamiento de la aplicación.

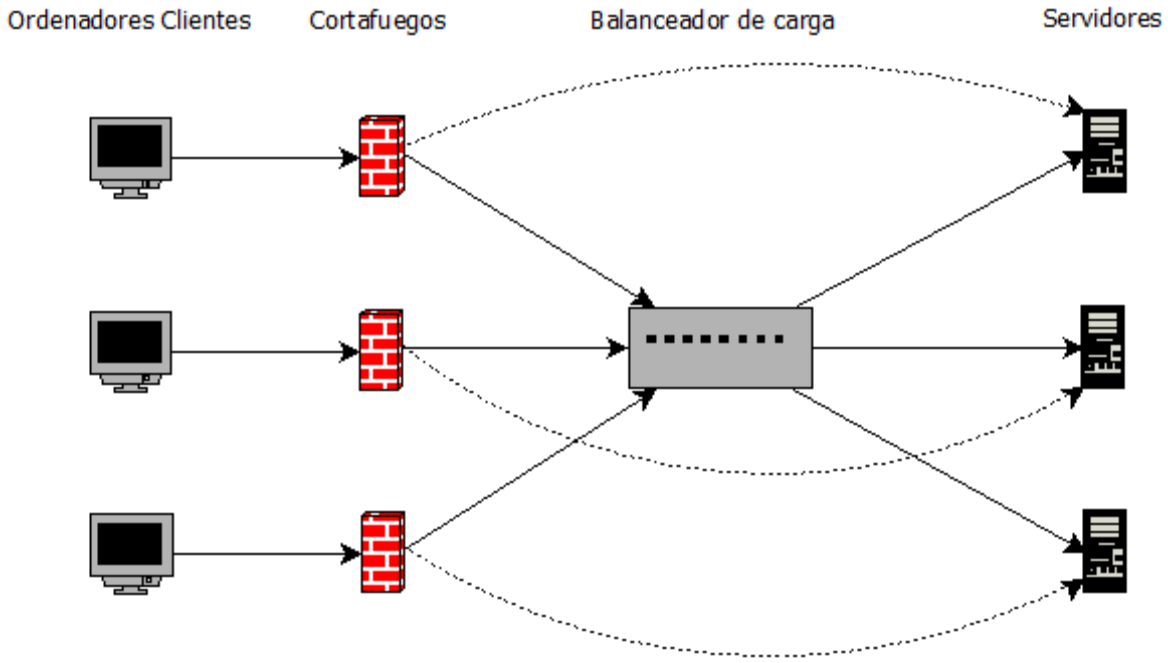


Imagen 57. Arquitectura Física

De esta imagen observamos:

- En primer lugar los clientes: ordenadores usados por los usuarios para acceder a la web y así poder jugar la partida.

- En segundo lugar los cortafuegos: usados para garantizar la seguridad e integridad del sistema. Saliendo de los cortafuegos observamos unas líneas discontinuas hasta los servidores sin pasar por el balanceador de carga, esto implica que el puerto en el que el servidor escucha las conexiones Websocket debe ser accesible al exterior. En términos de seguridad, sería aconsejable controlar que en el cortafuegos no existiese un cliente abriendo N-miles conexiones, ya que se produciría un ataque de denegación de servicio.

- En tercer lugar aparece el balanceador de carga: este balanceador asigna a cada cliente su servidor, este servidor devuelve una página web con código cliente (JavaScript) que abre una conexión directamente del cliente a él, de este modo, el balanceador de carga tiene que distribuir la carga de forma uniforme entre los servidores. Así, los jugadores juegan partidas unos contra otros en el mismo servidor, lo que facilita las migraciones de servidores. Esta distribución de los jugadores se realizaría o por Round Robin[6], o según el origen físico de la petición (GeoIP).

[6] Aprovisionamiento múltiple y redundante de servidores de servicios IP.

- En último lugar encontramos los servidores: en los servidores transcurre toda la lógica de negocio del sistema, dando el servicio requerido por los clientes.

5.2 – Diseño:

En este apartado mostraremos los distintos diagramas del proyecto.

5.2.1 – Diagramas de casos de uso:

En este caso, los diagramas de casos de uso de diseño son los mismos que se realizaron en la fase de análisis del proyecto, más concretamente en el Capítulo 4, Apartado 4.2, ya que con esa primera aproximación del modelo, no se requiere modificación alguna de dichos diagramas.

5.2.3 – Protocolo:

El protocolo elegido para realizar la conexión entre cliente y servidor en este proyecto es el HTTP, a través de esta conexión la información se intercambia a través de objetos JSON como se explicará posteriormente en esta documentación.

En la siguiente imagen podremos ver las direcciones de los mensajes que se realizan, quién las realiza, el tipo de mensaje y su contenido.

Objeto		Dirección		Mensajes	
Tipo	Contenido	Servidor	Cliente	Mensaje	Respuesta
register	Name (string)	←		regName	Res
card	Card (card)	→		Res	-
notification	Msg (binary)	→		Res	-
lineClaim	-	←		claimLine	lineClaimResult
lineClaimResult	Result (boolean)	→		lineClaimResult	
bingoClaim	-	←		claimBingo	bingoClaimResult
bingoClaimResult	Result (boolean)	→		bingoClaimResult	

Tabla 42. Comunicación protocolo HTTP

5.2.3 – Diagrama de clases:

A continuación vemos el diagrama de clases del proyecto. En él se muestra como se comunican las distintas clases entre sí y que funciones son invocadas por cada clase. En la cabecera de cada clase encontramos tres notaciones distintas entre paréntesis:

- (*) : indica que la clase modela múltiples procesos en la aplicación.

- (1) : indica que es un proceso único en la aplicación, los procesos únicos son identificables por el nombre de la clase.

- (0) : indica que la clase actúa como una librería de funciones.

Como explicamos en el Capítulo 1, los nombres de las funciones van seguidos de una barra "/" y un número que indica el número de argumentos de la función, su aridad.

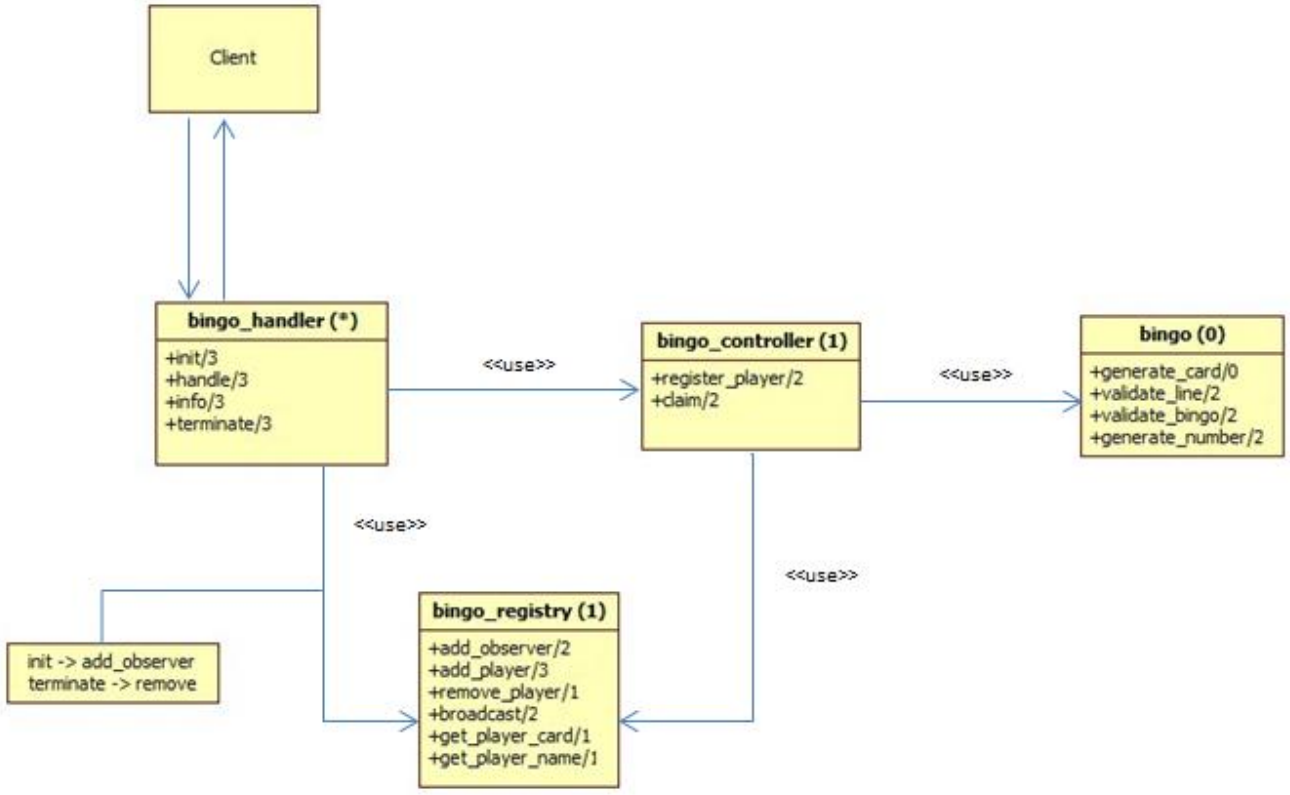


Imagen 58. Diagrama de clases

5.2.4 – Diseño tablas ETS (Erlang Term Storage):

Como en este proyecto se intenta mostrar el potencial de Erlang frente a otras tecnologías a la hora de crear sistemas online con alta concurrencia, se construyó todo el sistema utilizando las mínimas tecnologías posibles además de Erlang para no interferir en el resultado final.

Por esa razón no se necesita persistencia en el sistema, por eso no se utiliza un sistema de gestión de bases de datos relacional como podría ser MySQL si no que usamos una estructura de datos propia de Erlang, que nos proporciona la posibilidad de almacenar registros de forma local, esta estructura es una tabla ETS que a continuación se explica.

Una de las características de las tablas ETS es la capacidad de almacenar gran cantidad de datos con un tiempo de acceso siempre constante, ya que en los lenguajes funcionales el tiempo de acceso a la información suele ser función logarítmica. Otra característica es que pueden proveer al desarrollador de un modo de extraer, almacenar y tratar la información con los mecanismos propios de Erlang. Además para que el acceso fuese más rápido, las funciones para manejar las funcionalidades del módulo ets se encuentran en formato BIF.

En el caso de una hipotética ampliación del proyecto en el que en la lógica de negocio se manejase un intercambio económico entre los jugadores y el administrador, la opción de tablas ETS no sería una solución óptima, pero la actualización de nuestro sistema a otro sistema con una base de datos que nos ofrezca persistencia sería trivial pasando las tablas ETS a por ejemplo una base de datos Mnesia[7] (Sistema de base de datos distribuida ideal para sistemas que se necesiten en funcionamiento continuo 24horas 7 días de la semana con características de tiempo real).

- **Tipos de Tablas:**

Podemos encontrar cuatro tipos de tablas ETS dependiendo de los algoritmos empleados para la constitución de la tabla, su almacenaje y la extracción de datos:

- **Conjunto (set):** Es el tipo por defecto. A semejanza de los conjuntos como concepto matemático, cada elemento (cada clave de cada tupla) debe de ser único a la hora de realizar la inserción dentro del conjunto. El orden interno de los elementos no está definido.

- **Conjunto ordenado (ordered_set):** Es igual que el tipo anterior, pero en este caso los datos entrantes en el conjunto son ordenados mediante la comparación de su clave con las claves de los datos almacenados a través de los comparadores < y >, siendo el primer elemento el más pequeño.

- **Bolsa (bag):** La bolsa elimina la restricción de que el primer elemento ya exista, pero mantiene la propiedad de que las tuplas, comparadas en su conjunto con otras, deben de ser distintas.

[7] Un DBMS (Data Base Management System) para telecomunicaciones distribuidas (http://www.erlang.org/doc/man/mnesia.html).

- **Bolsa duplicada (duplicate_bag):** Igual que la anterior, pero eliminando la restricción de que las tuplas en su conjunto y comparadas con el resto deban de ser diferentes, es decir, se permiten tuplas repetidas (o duplicadas).

Dependiendo del tipo de datos que necesitemos almacenar en las tablas podremos elegir uno u otro tipo de tabla. Por ejemplo, si tenemos que almacenar términos de forma ordenada para su extracción podemos emplear un *ordered_set*, mientras que si la información que queremos almacenar puede llegar a repetirse podríamos optar por alguna de las bolsas, según el grado de repetición que queramos o tengamos que permitir para los datos.

- **Acceso a las ETS:**

Las tablas ETS son creadas por un proceso que puede hacerlo con opciones de accesibilidad que permitan su acceso por otros procesos o no. Los parámetros de seguridad que podemos emplear para garantizar el acceso o denegarlo, según el caso, son los siguientes:

- **Private:** Crea la ETS de ámbito privado. Esto quiere decir que no permite a ningún otro proceso el acceso a la misma.

- **Protected:** El ámbito protegido para la ETS garantiza el acceso de lectura a todos los procesos que conozcan el identificador de la ETS, pero sólo permite la escritura para el proceso que la creó.

- **Public:** Garantiza el acceso a todos los procesos, tanto para lectura como escritura, a la ETS a través del identificador de la misma.

- **Creación de una ETS:**

Para crear una tabla ETS emplearemos la función new/2 del módulo ***ets*** cuyos parámetros son el nombre de la tabla (un átomo) y las opciones para la creación de la misma.

Las opciones están en formato de lista. Cada elemento de la lista corresponderá a cada una de las siguientes secciones:

- **Tipo de tabla:** Se debe de especificar alguno de los tipos de ETS vistos: *set*, *ordered_set*, *bag* o *duplicate_bag*.

- **Acceso a la tabla:** Se debe de especificar alguno de los tipos de accesos para la ETS vistos: *public, protected* o *private*.

- **Named_table:** Si se especifica esta opción, el primer parámetro de la función es empleado como identificador para poder acceder a la tabla.

- **Keypos:** En caso de que queramos que la clave de la ETS no sea el primer elemento de la tupla podemos agregar esta opción de la forma: {keypos, Pos}. Siendo “Pos” un número entero dentro del rango de elementos de la tupla.

- **Heir:** El sistema puede establecer un proceso hijo al que pasarle el control de la ETS, de modo que si algo le sucediese al proceso que creó la ETS, el proceso hijo recibiría el mensaje:

  ```
  {'ETS-TRANSFER', id, FromPid, HeirData}
  ```

Y tomaría en propiedad la tabla. La configuración sería:

```
{heir, Pid, HeirData}
```

En caso de no especificar un heredero, si el proceso propietario de la tabla termina su ejecución la tabla desaparece con él.

- **Concurrencia:** Por defecto, las ETS mantienen un nivel de concurrencia por bloqueo completo, es decir, mientras se está trabajando con la tabla ningún otro proceso puede acceder a ella, ya sea para leer o escribir. No obstante, a través de la opción:

  ```
  {read_concurrency, true}
  ```

Activamos la concurrencia de lectura. Esta opción es buena si el número de lecturas es mayor que el de escrituras, ya que el sistema adapta internamente los datos para que las lecturas puedan emplear incluso los diferentes procesadores que pueda tener la máquina.

Para activar la concurrencia en la escritura, se debe emplear la opción siguiente:

```
{write_concurrency, true}
```

La activación de la escritura sigue garantizando tanto la atomicidad como el aislamiento. Esta opción está recomendada si el nivel de concurrencia de lectura/escritura de los datos almacenados en la ETS provocan excesivo tiempo de espera y fallos a consecuencia de este cuello de botella. La nota negativa, vuelve a ser la penalización existente al realizar las escrituras concurrentes.

- **Compressed:** Los datos de la ETS se comprimen para almacenarse en memoria. Al trabajar sobre datos comprimidos los procesos de búsqueda y toma de datos son más costosos. Esta opción es aconsejable cuando sea más importante el consumo de memoria que la velocidad de acceso.

A continuación se muestra como se crea nuestra tabla ETS en el proyecto:

```
init([]) ->
    ets:new(?ETS, [named_table, public, set, {read_concurrency,true},
    {write_concurrency, true}, {keypos, #player.connection}]),
    {ok, #state{}}.
```

Imagen 59. Crear tabla ETS

Como se ve en la imagen del código, se escogen la opción “set” ya que no nos importa el orden en el que se introducen los elementos en la tabla y los identificamos por su pid como clave.

En cuando al acceso se escoge la opción “public” ya que nos interesa que se pueda leer y escribir en la tabla a través de cualquier proceso que conozca su pid.

Para trabajar con la tabla vemos como activamos la concurrencia de lectura y escritura mediante “read_concurrency, true” y “write_concurrency, true” respectivamente.

Por último, como clave elegimos el pid del jugador escribiendo “{keypos, #player.connection}”

- **Lectura y escritura en ETS:**

Una vez que tenemos creada una ETS podemos comenzar a trabajar con ella. Para agregar elementos podemos emplear la función insert/2 cuyo primer parámetro es el identificador de la tabla (o su nombre en caso de *named_table*), siendo el segundo parámetro un término o una lista de términos para su inserción.

```
-spec add_observer(pid(), binary()) -> ok.
add_observer(Conn, UA) ->
    true = ets:insert(?ETS, [#player{connection=Conn, browser=UA,
                                     card=undefined}]),
    ok.
```

Imagen 60. Lectura y escritura tabla ETS

En la imagen se muestra como añadimos un observador a la tabla ETS, poniendo la macro “?ETS” que en Erlang toma siempre el valor del nombre del módulo y a continuación como argumento escribimos el registro “#player” con los campos y el valor de los campos (los campos con valor sin especificar se ponen undefined por defecto).

Un observador, si se registra con un nombre de jugador, se convertirá en un jugador y adquirirá un cartón de juego para comenzar a jugar, así como el nombre elegido. Como su pid ya estaba registrado en la tabla, sólo hay que buscarlo y actualizar los valores del observador, para ello se escribió el siguiente código.

```
-spec add_player(pid(), binary(), bingo:card()) -> non_neg_integer().
add_player(Conn, DisplayName, Card) ->
    true = ets:update_element(
             ?ETS, Conn, [{#player.card, Card}, {#player.name, DisplayName}]),
```

Imagen 61. Actualizar registros en la tabla ETS

En este código se observa como mediante “ets:update_element” buscando en la tabla de nombre “?ETS” por el pid del proceso “Conn” se actualizan los campos “card” y “name” del registro “#player” dándoles el valor “Card” y “DisplayName”.

Para leer un elemento de la tabla, usamos “lookup” del modo que mostramos en la imagen donde se ve el código usado para obtener en la primera cláusula el cartón del jugador y en la segunda el nombre del jugador.

```
-spec get_player_card(pid()) -> bingo:card().
get_player_card(Conn) ->
        [InfoPlayer] = ets:lookup(?ETS,Conn),
        InfoPlayer#player.card.

-spec get_player_name(pid()) -> binary().
get_player_name(Conn) ->
        [InfoPlayer] = ets:lookup(?ETS,Conn),
        InfoPlayer#player.name.
```

Imagen 62. Leer registro ETS

- **Búsqueda avanzada “ets:select”:**

En las tablas ETS hay varias funciones BIF para realizar búsquedas avanzadas, dos de las más utilizadas son “match/1, match/2 y match/3” y “ select/1, select/2 y select/3”.

La teoría de la concordancia para las ETS se puede emplear tanto para funciones select/2, como para las funciones match/2. Esta concordancia se basa en pasarle la información al núcleo de ETS para realizar la extracción. Requiere que se puedan identificar variables como tal o bien con el uso de comodines.

Para esto se definen dos átomos que tienen una semántica especial para el gestor de las ETS. Son las llamadas variables *sin importancia (don´t care)* y el comodín. Las variables sin importancia se pueden especificar como *'$0', '$1', '$2'...* La numeración sólo es relevante en caso de especificar la forma en la que se obtendrán los resultados.

El otro tipo de átomo con significado específico para las ETS es el comodín '_'. Ya se vió en el Capítulo 2 que el signo de subrayado se utiliza para indicar que el dato en esa posición no interesa.

En el proyecto se utiliza select/2 ya que da la posibilidad no sólo de enviar una tupla de concordancia, sino también una parte de guardas y la proyección (el cómo se visualizarán en el resultado). Se usa en dos casos select/2:

- Primer caso:

```
length(ets:select(?ETS, [{#player{card='$1', _='_'}, [{'/=', '$1', undefined}], [true]}])).
```

Imagen 63. Búsqueda avanzada por cartón

En esta línea de código lo que se realiza una búsqueda avanzada usando “ets:select”, el primer argumento es el nombre de la tabla mediante la macro “?ETS”, a continuación se busca en el registro “#player” aquellos jugadores que tiene un cartón con un valor distinto a “undefinied”

```
-spec broadcast(Type::atom(), Value::atom()) -> ok.
broadcast(Type, Value) ->
    Pids = ets:select(?ETS,[{#player{connection='$1', _='_'}, [], ['$1']}]),
    [Pid ! {msg, {Type, Value}} || Pid <- Pids].
```

Imagen 64. Búsqueda avanzada por Pid

En esta línea de código se escogen todos los jugadores con el campo "connection" del registro #player para obtener su valor y enviar posteriormente un mensaje.

5.3 – Diseño de las interfaces:

5.3.1 – Componentes de las interfaces:

Antes de comenzar con el diseño de las interfaces, es necesario explicar que la interfaz del servidor no es más que la consola Erlang previamente explicada en los primeros capítulos de la documentación. De este modo, procedemos a explicar los componentes de la página web cliente de nuestro sistema.

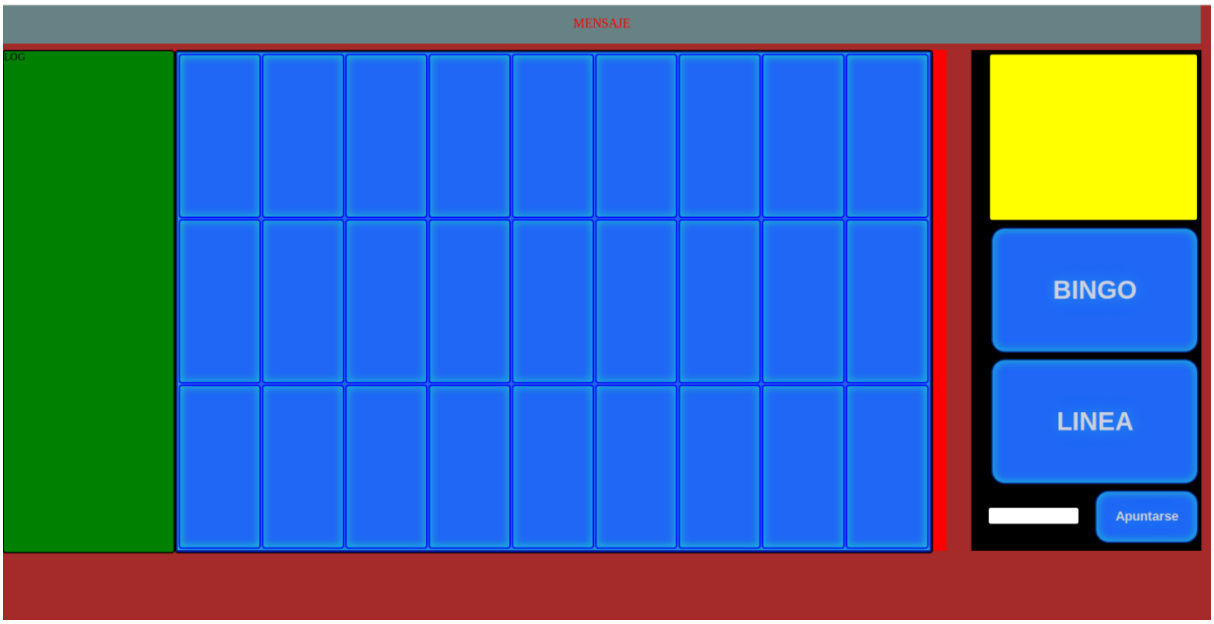


Imagen 65. Interfaz cliente

Como podemos ver en la anterior imagen, disponemos de una interfaz diseñada en HTML y CSS.

El diseño se divide en 3 div principales contenidos dentro el div "global" coloreado con color marrón. Estos 3 divs son:

- "header": en la parte superior de la ventana, donde se escribirá el título del juego, coloreado en gris.

- "right": al lado derecho de la ventana, coloreado en negro, contiene:

- Div “display”: donde se mostrarán los números de la partida (coloreado en amarillo).
- Button “bingoButton”: botón utilizado para cantar bingo por el jugador.
- Button “lineButton”: botón utilizado para cantar línea por el jugador.
- Input “textBox”: campo de texto utilizado para introducir el nombre con el que se registra el jugador.
- Button “regButton”: botón utilizado para registrar al jugador en la partida con el nombre introducido en “textBox”.

- “left” : coloreado en rojo, se extiende desde el lado izquierdo de la pantalla hasta el div “right”, contiene el div “log” de la partida coloreado en verde y el div “card” donde se sitúa la tabla “cardTable” que se rellena con los números de la partida.

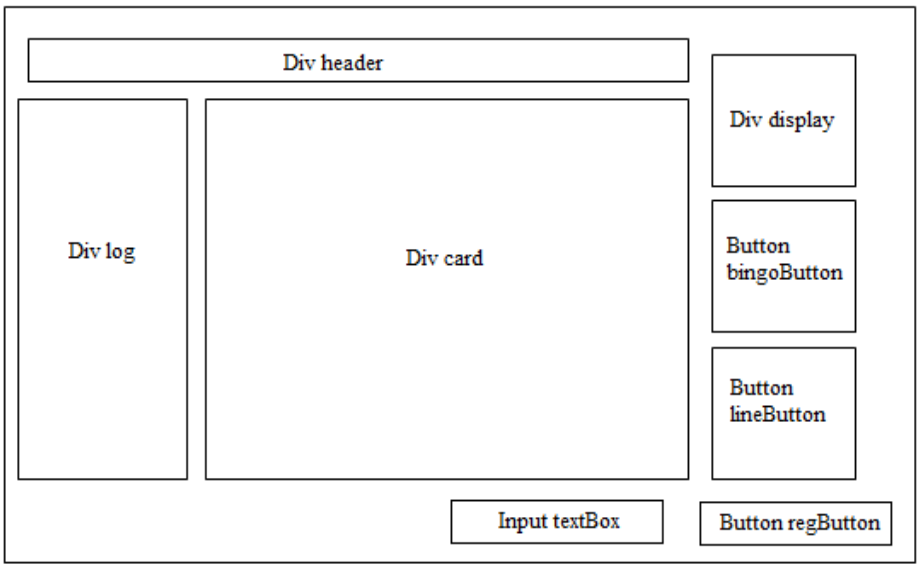
IFG–01	Aplicación Web
Descripción	*Interfaz que muestra laa opciones que tiene el usuario de interactuar con la aplicación y los elementos que controlará el sistema para desarrollar la lógica del juego.*
Activación	*Al escribir la url en el navegador.*
Boceto	
Eventos	*Onclick claimBingo()* *Onclick claimLine()* *Onclick regName()*

Tabla 43. Interfaz gráfica aplicación web

CAPÍTULO 6

PRUEBAS

6.1 - Pruebas de caja negra:

Conociendo una función específica para la que fue diseñado el proyecto, se pueden diseñar pruebas que demuestren que dicha función está bien realizada. Dichas pruebas son llevadas a cabo sobre la interfaz del software, es decir, de la función, actuando sobre ella como una caja negra, proporcionando unas entradas y estudiando las salidas para ver si concuerdan con las esperadas.

A continuación se presentan las tablas de las pruebas realizadas en el juego:

CP - 01	Registrarse como jugador
Tipo de Prueba	*Caja negra.*
Objetivo	*Obtener un nombre de jugador y un cartón para jugar la partida.*
Procedimiento de Prueba	*Se rellena el campo de texto y se hace click sobre el botón "Apuntarse".*
Salida Esperada	*La aplicación debería registras al jugador con el nombre introducido en el campo de texto y servirle un cartón para iniciar la partida.*
Salida Obtenida	*La aplicación registra al jugador con el nombre adecuado y le sirve un cartón para iniciar la partida.*

Tabla 44. CP-01 Registrarse como jugador

CP - 02	Reclamar línea por primera vez
Tipo de Prueba	*Caja negra.*
Objetivo	*Reclamar una línea durante la partida.*
Procedimiento de Prueba	*Se pulsa el botón "Linea" cuando se obtienen todos los números de una de las tres filas del cartón.*
Salida Esperada	*La aplicación debería informarnos a través del log y la cabecera que la línea ha sido válida.*
Salida Obtenida	*La aplicación informa a través del log y la cabecera que la línea ha sido valida.*

Tabla 45. CP-02 Reclamar línea por primera vez

CP - 03	Reclamar línea después de una línea válida
Tipo de Prueba	*Caja negra.*
Objetivo	*Reclamar una línea durante la partida y ver el resultado*
Procedimiento de Prueba	*Se pulsa el botón "Linea" una vez ya se haya reclamado línea con éxito por el jugador o cualquier otro jugador.*
Salida Esperada	*La aplicación debería informar a través del log y de la cabecera que la línea es inválida.*
Salida Obtenida	*La aplicación informa a través del log y la cabecera que la línea es inválida.*

Tabla 46. CP-03 Reclamar línea después de una línea válida

CP - 04	Reclamar bingo válido
Tipo de Prueba	*Caja negra.*
Objetivo	*Reclamar un bingo válido durante la partida.*
Procedimiento de Prueba	*Se pulsa el botón "Bingo" cuando se obtienen todos los números del cartón.*
Salida Esperada	*La aplicación debería informarnos a través del log y la cabecera que el bingo ha sido válido.*
Salida Obtenida	*La aplicación informa a través del log y la cabecera que el bingo ha sido válido.*

Tabla 47. CP-04 Reclamar bingo válido

CP - 05	Reclamar bingo inválido
Tipo de Prueba	*Caja negra.*
Objetivo	*Reclamar un bingo inválido durante la partida.*
Procedimiento de Prueba	*Se pulsa el botón "Bingo" sin haber salido todos los números del cartón..*
Salida Esperada	*La aplicación debería informarnos a través del log y la cabecera que el bingo no ha sido válido.*
Salida Obtenida	*La aplicación informa a través del log y la cabecera que el bingo no ha sido válido.*

Tabla 48. CP-05 Reclamar bingo inválido

6.2 - Pruebas de caja blanca:

Las pruebas de caja blanca se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente. Se escogen valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa o función y así cerciorarse de que se devuelven los valores de salida adecuados.

- Caja blanca 1: Engloba las funciones de identificación de un jugador.

1
```
regName = function(){
    var name = document.getElementById("textbox").value;
    if (validateName(name)){
        connection.send(JSON.stringify({type: "register", content: name}));
```
2
```
        }else{
            document.getElementById('textbox').focus();
        }
};
```
3
```
validateName = function (playerName) {
   playerName = document.getElementById("textbox").value;
        if (playerName == null || playerName == "") {
            alert("Introduce un nombre de registro");
```
4
```
        return false;
        }else{
            return true;
        }
},
```
5
```
<<"register">> ->
    DisplayName = proplists:get_value(<<"content">>, Data),
    case bingo_controller:register_player(self(), DisplayName) of
        {ok, Card} ->
            Res = jiffy:encode({[{type, <<"card">>}, {content, bingo:card2ison(Card)}]}),
            {reply, {text, Res}, Req, State};
```
6
```
        {error, wait_for_next_game} ->
            Msg = list_to_binary(io_lib:format(
                "Error: registro inhabilitado hasta el final de la partida-n", [])),
            Res = jiffy:encode({[{type, <<"notification">>}, {content, Msg}]}),
            {reply, {text, Res}, Req, State}
    end
```
7
```
waiting_for_players({register, Conn, PlayerId}, _From, #state{min_players=Min}=State) ->
    Card = bingo:generate_card(),
    bingo_registry:broadcast(
        notification, list_to_binary(
            io_lib:format("'~s' se ha unido a la partida", [PlayerId]))),
    case bingo_registry:add_player(Conn, PlayerId, Card) >= Min of
        true ->
            gen_fsm:send_event(?MODULE, start_countdown),
            {reply, {ok, Card}, countdown, State};
```
8
```
        false ->
            {reply, {ok, Card}, waiting_for_players, State}
    end.

countdown({register, Conn, PlayerId}, _From, State) ->
    Card = bingo:generate_card(),
    bingo_registry:add_player(Conn, PlayerId, Card),
    bingo_registry:broadcast(
        notification, list_to_binary(
            io_lib:format("'~s' se ha unido a la partida", [PlayerId]))),
    {reply, {ok, Card}, countdown, State}.
```

Imagen 66. Código de Caja Blanca 1

- Diagrama de flujo de control del método:

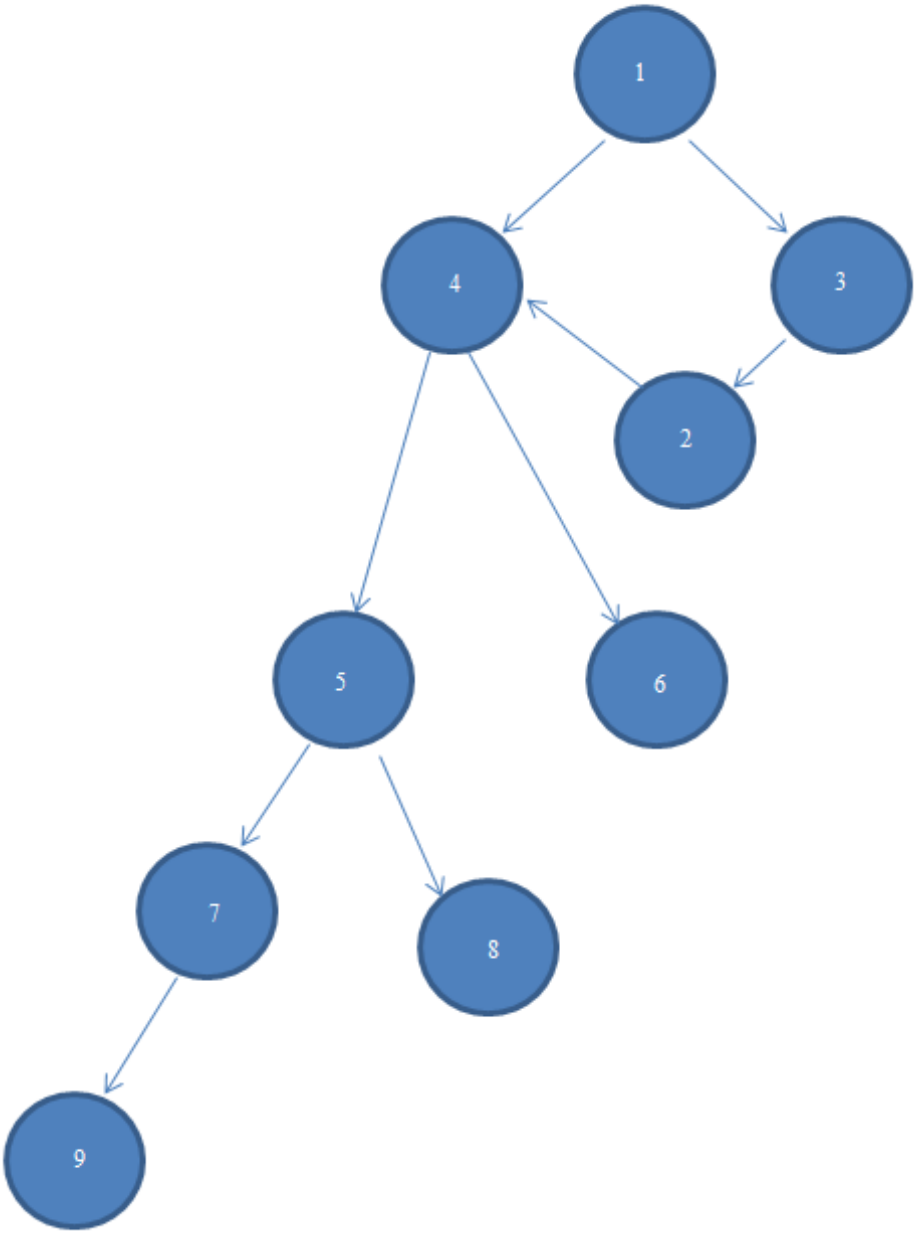


Imagen 67. Diagrama de flujo de Caja Blanca 1

- Diseño y realización de pruebas:

La complejidad ciclomática del sistema es 6, por lo que se diseñarán seis caminos de pruebas que serán los siguientes:

Camino 1: 1, 3, 2, 4, 6
Camino 2: 1, 3, 2, 4, 5, 8
Camino 3: 1, 3, 2, 4, 5, 7, 9
Camino 4: 1, 4, 6
Camino 5: 1, 4, 5, 8
Camino 6: 1, 4, 5, 7, 9

Prueba 1	Camino 1
Usuario	*(blanco)*
Salida Obtenida	*Mensaje de error indicando que el registro está inhabilitado hasta el final de la partida.*
Comentarios	*El usuario se registra bien después de tener el mensaje de alerta por el cual no puede registrarse con el campo de texto vacío.*

Tabla 49. Camino 1

Prueba 2	Camino 2
Usuario	*(blanco) y después registro con nombre Anibal*
Salida Obtenida	*El jugador Anibal obtiene el cartón para jugar.*
Comentarios	*El usuario adquiere el cartón de juego pero deberá esperar a que se registren N jugadores para empezar la partida, la máquina de estados permanece en estado "waiting_for_players".*

Tabla 50. Camino 2

Prueba 3	Camino 3
Usuario	*(blanco) y después registro con nombre Anibal*
Salida Obtenida	*El jugador Anibal obtiene el cartón para jugar. Se notifica en el log que el jugador Anibal se ha unido a la partida.*
Comentarios	*El usuario adquiere el cartón de juego y comienza la partida.*

Tabla 51. Camino 3

Prueba 4	Camino 4
Usuario	*Anibal*
Salida Obtenida	*Mensaje de error indicando que el registro está inhabilitado hasta el final de la partida.*
Comentarios	*El usuario se registra bien pero debe esperar que comience una nueva partida para jugar*

Tabla 52. Camino 4

Prueba 5	Camino 5
Usuario	*(Anibal*
Salida Obtenida	*El jugador Anibal obtiene el cartón para jugar.*
Comentarios	*El usuario adquiere el cartón de juego pero deberá esperar a que se registren N jugadores para empezar la partida, la máquina de estados permanece en estado "waiting_for_players".*

Tabla 53. Camino 5

Prueba 6	Camino 6
Usuario	*Anibal*
Salida Obtenida	*El jugador Anibal obtiene el cartón para jugar. Se notifica en el log que el jugador Anibal se ha unido a la partida.*
Comentarios	*El usuario adquiere el cartón de juego y comienza la partida.*

Tabla 54. Camino 6

6.3 – Pruebas de carga:

6.3.1 – Propósito:

El rendimiento de la aplicación se basa en el modelado de cada cliente (jugador con el navegador conectado al servidor) como un proceso en la máquina virtual Erlang (bingo_handler). En este capítulo se analizarán las principales claves para el rendimiento del bingo en base a lo anteriormente explicado.

6.3.2 – Consideraciones:

Para la realización de unas pruebas de carga con unos resultados realistas es obligatorio el uso de una máquina o cluster de máquinas cliente separados de la máquina servidor. Estas máquinas cliente se encargarían de la generación de carga y de la recogida de las mediciones, limitándose la máquina servidor al trabajo real que haría en un despliegue en producción: servir partidas.

El uso de la misma máquina para medir, generar tráfico y servir tráfico obviamente desvirtúa muchos de los valores medidos, en particular aquellos que dependen de la carga de la máquina servidor (como los tiempos de respuesta), ocupada doblemente generando peticiones y midiendo tratando de limitarse a procesar peticiones. Si son válidos, sin embargo, los datos relacionados al número de clientes concurrentes soportados por el servidor: en tal caso el servidor, estando sobrecargado con otras tareas, será quien pueda soportar un menor número de clientes en estas pruebas del que soportaría en un despliegue real.

6.3.3 – Análisis de los puntos críticos de rendimiento:

- **Número de sockets TCP que puede abrir el servidor:** se pueden monitorizar cada segundo usando '`$ watch lsof -ni:8081 -Stp:established | wc -|`'. Para conseguir un número elevado en un sistema Linux hay que configurar la máquina de manera adecuada (véase http://www.lognormal.com/blog/2012/09/27/linux-tcpip-tuning/). En un portátil medio es posible mantener entre 10.000 y 100.000 conexiones TCP concurrentes.

- **Número de procesos concurrentes que pueden correr en una máquina virtual de Erlang:** la máquina virtual Erlang acepta un número máximo de procesos concurrentes de 262144 por defecto. Este número se puede incrementar hasta 3 órdenes de magnitud usando el flag +P.
  El límite de procesos, en la práctica, depende principalmente del procesador. Si bien cada proceso consume memoria RAM, los procesos en la máquina virtual Erlang son tan ligeros que el factor crítico pasa a ser la CPU (núcleos y frecuencia) disponible para coordinarlos y ejecutarlos todos. En un portátil medio no es complicado ejecutar benchmarks con más de un millón de procesos concurrentes. Para una aplicación como el bingo, cientos de miles es una cifra realista.

- **Overhead del protocolo JSON de la aplicación bingo sumado al overhead del protocolo Websocket respecto de TCP:** esto influye en el tiempo de codificación y decodificación de cada mensaje entre cliente y el servidor. Cuantos mas ciclos de CPU se necesiten, menos CPU queda libre para garantizar la concurrencia del resto de mensajes. A pesar que JSON no es un protocolo óptimo para obtener un óptimo rendimiento (está basado en texto; protocolos como Google Protocol Buffers cumplen mejor esta misión), si es el más universal y de ahí su elección.
  Tanto JSON como el propio protocolo Websocket se añaden overhead a la conexión TCP. Son, sin embargo, mucho mas ligeros de lo que es una conexión tradicional AJAX (basada en HTTP sobre TCP). El Websocket añade a esta optimización en el tamaño del mensaje el beneficio de la comunicación en el lado servidor.

- **Errores de programación en la aplicación bingo que consuman memoria, tiempo o CPU extra, lo que repercute directamente en 2:** siendo esta aplicación bingo parte de un estudio sobre Erlang/OTP, ciertamente contiene algún error de programación o prácticas no óptimas.

6.3.4- Procedimiento y conclusiones:

Se intentó el uso de la aplicación Thor-hammer (https://www.npmjs.org/package/thor-hammer, basada en Node.js) para la medición del número máximo de jugadores concurrentes que un servidor de bingo podría soportar. Utilizando esta herramienta, se probó a abrir 100.000 conexiones concurrentes al servidor de bingo.

rico@phx: ~/TFG/bingo$ Thor-hammer "ws://localhost:8081/bingo/connect" –W 4 –A 100000

Thor:

God of Thunder, son of Odin and smasher of Websockets!

Thou shall:
- Spawn 4 workers.
- Create all the concurrent/parallel connections.
- Smash 100000 connections with the mighty Mjölnir.

The answers you seek shall be yours, once I claim what is mine.

Connecting to ws://localhost:8081/bingo/connect

Progress :: Created 52811, Active 178410146

Online	490060 milliseconds
Time taken	490064 milliseconds
Connected	100000
Disconected	0
Failed	0
Total transfered	153.54 MB
Total received	12.68MB

Durations (ms)	Min	Mean	Sttdev	Median	Max
Handshaking	290	63424	23591	66038	101603
Latency	NaN	NaN	NaN	NaN	NaN

Tabla 55. Tabla de duración de tiempos

Percentile (ms)	50%	66%	75%	80%	90%	95%	98%	98%	100%
Handshaking	66038	72907	82669	87921	93739	95795	97884	98481	101603
Latency	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabla 56. Tabla de percentiles

La aplicación acabó con éxito en un tiempo inferior a 4 minutos. Los números de conexiones TCP abiertas y el número de procesos en la máquina virtual Erlang durante la ejecución del test no se acercó en ningún momento a los 100.000 esperados. Esto implica que Thor-hammer no fue capaz de producir una prueba válida y por lo tanto los resultados no son del todo confiables.

Como aproximación, basádonse en los parámetros explicados con anterioridad, es seguro estimar que el bingo es capaz de soportar entre 10.000 y 100.000 jugadores concurrentes en un servidor (después de realizar pequeñas modificaciones en el sistema host Linux para incrementar el número máximo de conexiones y disminuir el TIME_WAIT de cada una).

CAPÍTULO 7

MANUALES

Capítulo 7 - Manuales

7.1 - Introducción:

A lo largo de este tema, explicaremos las distintas opciones que tiene el administrador para instalar y controlar el sistema y cómo puede jugar un usuario a nuestro bingo.

7.1.1 - Manual de administración:

En primer lugar, tras instalar Erlang como se muestra en el Capítulo 2 de la documentación, se debe instalar "rebar", para esto hay que ejecutar en la consola la orden apt-get install erlang rebar siendo superusuario.

Para hacerse con el código del proyecto, el administrador usará Github.com donde tiene dos maneras de hacerlo posible.

La primera es clonar el proyecto del repositorio remoto https://github.com/trazzo/bingo.git, para ello el administrador deberá crear un directorio donde quiere alojar el código del proyecto y a continuación ejecutar git init para hacer de esa carpeta su propio repositorio loca y a continuación ejecutar la orden git clone https://github.com/trazzo/bingo.git. De este modo, Github.com nos copiará todo el código del proyecto en nuestro repositorio local.

La segunda manera es entrar directamente en https://github.com/trazzo/bingo.git y descargar el archivo .zip que contiene los archivos del proyecto, descomprimirlo y extraerlo donde él quiera.

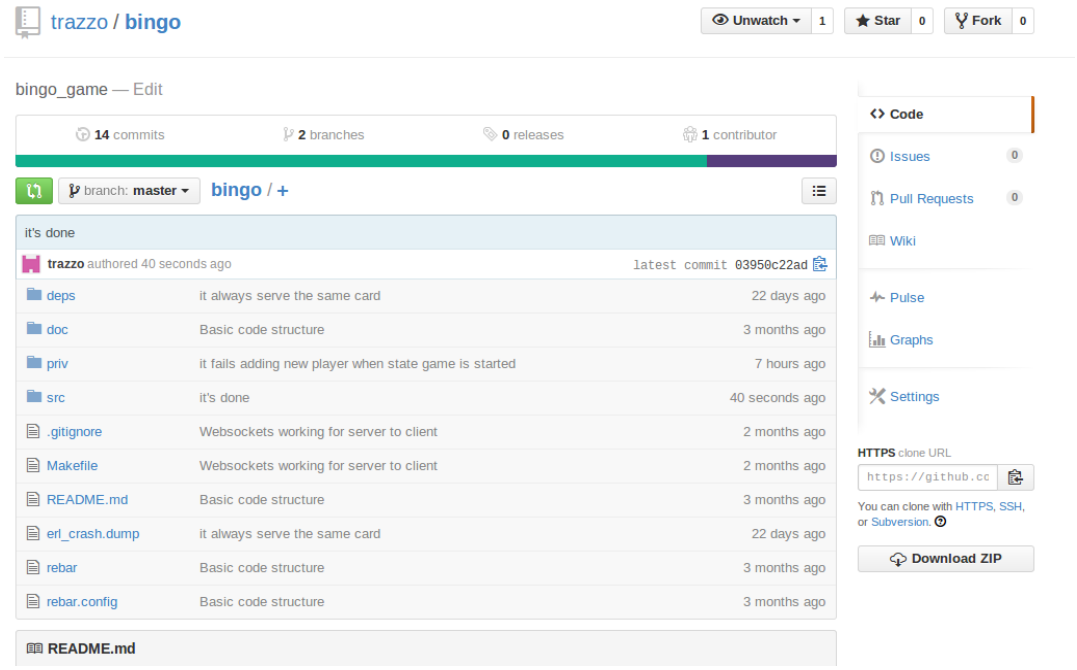


Imagen 68. GitHub

Ya con el código en el sistema, el administrador deberá compilar el codigo desde la consola accediendo desde la consola al directorio que escogió para guardar los archivos y ejecutar make. Con el código compilado, para hacer funcionar nuestro servidor ejecutaremos desde una nueva consola el comando make go. Una vez ejecutemos este comando, comprobaremos como en nuestra segunda consola se abre automáticamente el shell Erlang.

Teniendo el shell de Erlang listo para usar, el administrador puede usar los diferentes comandos que se detallaron en el Capítulo 2 de esta documentación.

Otros comandos muy útiles para administrar el servidor se explican a continuación:

- ets:tab2list(bingo_registry).

Sirve para ver como una lista la tabla ETS. En dicha lista podremos ver los observadores de la partida y los usuarios registrados como jugadores en el sistema, con qué navegador accedieron a la aplicación, su Pid y el nombre de registro de jugador y su cartón asociado (en caso de no ser un observador)

En este ejemplo vemos la lista de la tabla ETS que utilizamos en el módulo "bingo_registry", en la tabla se observa un jugador registrado con el nombre Quique y un observador sin nombre.

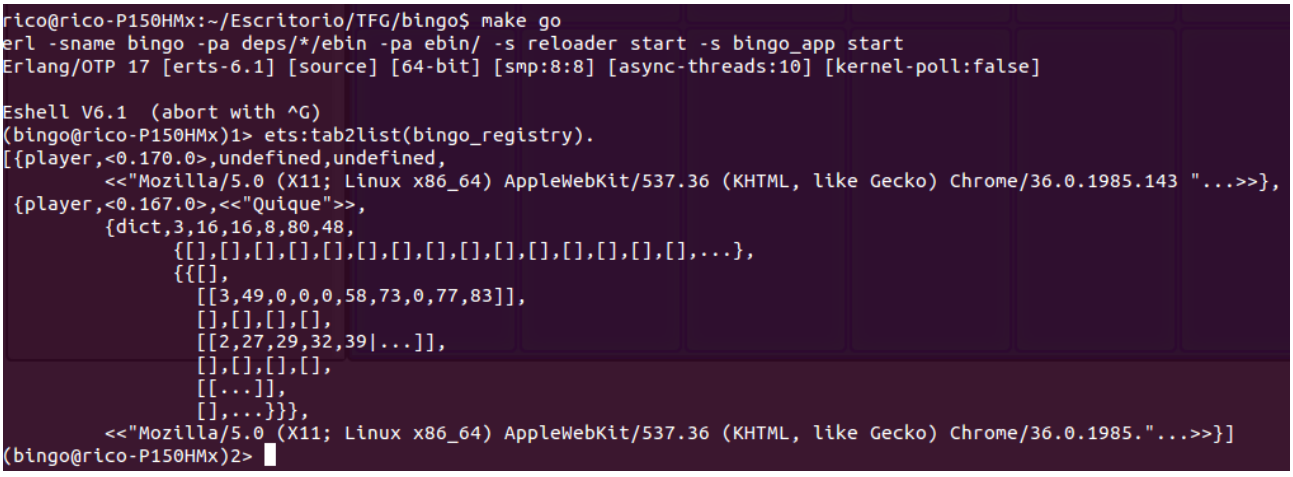


Imagen 69. Resultados tabla ETS

En la imagen de código se aprecia un registro con pid <0.170.0>, con el campo nombre y cartón "undefinied" lo que nos indica que se trata de un observador y con el campo que nos indica el navegador que está usando.

El siguiente registro tiene un pid <0.167.0>, registrado como dijimos antes con el nombre "Quique" , con un cartón asignado que es nuestro diccionario y con el campo que nos indica el navegador que está usando.

- Sys:get_state(bingo_controller).

Este comando sirve para ver el estado de nuestra máquina de estados finitas, es decir, el módulo “bingo_controller”.

Siguiendo con el ejemplo, en el que sólo hay un jugador registrado y un observador en la partida, el estado debería ser “waiting_for_players” ya que el mínimo de jugadores para iniciar la partida configurado es 3, de modo que a continuación se verá la información completa que nos aporta este comando.

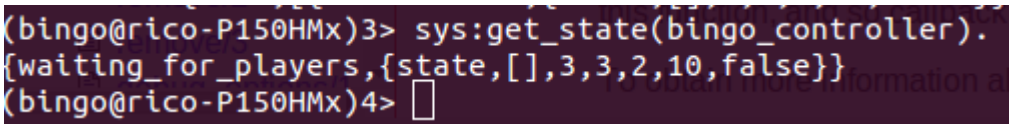


Imagen 70. Sys:get_state(bingo_controller)

En esta imagen apreciamos el estado de nuestra máquina “waiting_for_palyers” y los campos del registro estado, tanto los configurables (se explican posteriormente) como los que no.

- La lista vacía corresponde a la lista de números que salen aleatoriamente en el “bombo”, como se está esperando a más jugadores para iniciar la partida, no ha podido salir ningún número.
- El primer “3” pertenece al mínimo de jugadores necesarios para comenzar una partida.
- El segundo “3” indica los segundos que durará la cuenta atrás una vez que se cumpla el mínimo de jugadores requerido.
- El “2” son los segundos que pasarán entre número y número a la hora de salir del “bombo”.
- El “10” es la duración en segundos cuando finaliza una partida.
- “false” es la bandera que controla el que se haya cantado una línea o no, como obviamente no se ha podido cantar ninguna porque no se empezó la partida se mantiene ese valor.

– Erlang:process_info(pid(, , ,))

El comando process_info aporta toda la información necesaria sobre un proceso, se destaca entre esta información:

- "links", muestra los pids de los procesos a los que está ligado.
- "trap_exit", indica si el proceso admite excepciones.
- "messages", muestra los mensajes que no han sido procesados todavía por el proceso.

```
(bingo@rico-P150HMx)4> erlang:process_info(pid(0,167,0)).
[{current_function,{cowboy_websocket,handler_loop,4}},
 {initial_call,{cowboy_protocol,init,4}},
 {status,waiting},
 {message_queue_len,0},
 {messages,[]},
 {links,[<0.59.0>,#Port<0.1548>]},
 {dictionary,[]},
 {trap_exit,false},
 {error_handler,error_handler},
 {priority,normal},
 {group_leader,<0.45.0>},
 {total_heap_size,1220},
 {heap_size,610},
 {stack_size,14},
 {reductions,1804},
 {garbage_collection,[{min_bin_vheap_size,46422},
                      {min_heap_size,233},
                      {fullsweep_after,65535},
                      {minor_gcs,8}]},
 {suspending,[]}]
(bingo@rico-P150HMx)5>
```

Imagen 71. Process_info

– rr(Module).

El comando rr(Module) nos muestra los registros del módulo que se pasa por argumento, a continuación se muestran los registros de bingo_registry y de bingo_controller.

```
(bingo@rico-P150HMx)5> rr(bingo_controller).
[state]
(bingo@rico-P150HMx)6> rr(bingo_registry).
[player,state]
(bingo@rico-P150HMx)7>
```

Imagen 72. rr(Module)

7.1.2 – Configuración de la aplicación.

A la hora de escribir make go desde la consola para arrancar la aplicación, podemos hacer make go EXTRA_ARGS=”bingo.config” , de este modo se la aplicación lee los valores desde la máquina virtual Erlang. En el caso de que no se encontrase ningún archivo de configuración, la aplicación funcionaría con los valores establecidos por defecto en el módulo bingo en la función get_config.

- **Archivo de configuración.**

Un archivo de configuración es una proplist con una tupla por aplicación, en cada tupla la clave es el nombre de la aplicación y el valor es otra proplist que tiene como clave el nombre del campo a configurar y como valor los valores que quieran aplicarse.

En este caso se crea en el directorio priv/bingo_config un archivo bingo.config con el siguiente código:

```
[{bingo, [
  {min_players, 3}, %% El numero minimo de jugadores para empezar una partida
  {countdown, 5}, %% Duracion de la cuenta atras para el principio de la
  %% partida en segundos
  {time_between_numbers, 3} %% Segundos entre numero y numero
  {game_over_duration, 10} %% Segundos entre que finaliza una partida y
  %% empieza una nueva
]}].
```

Imagen 73 Archivo de configuración

7.2 - Manual de usuario:

Una vez que entre en la web del juego, un usuario puede observar la partida que está transcurriendo y las posteriores sin necesidad de jugar. Si desea jugar debe esperar a que finalice la partida y registrarse con un nombre de jugador escribiendo su nombre en el campo de texto inferior y haciendo click en el botón “Apuntarse”.

Imagen 74. Botón Apuntarse

Una vez registrado, recibirá un cartón con los números de la partida y esta empezará una vez acabe la cuenta atrás.

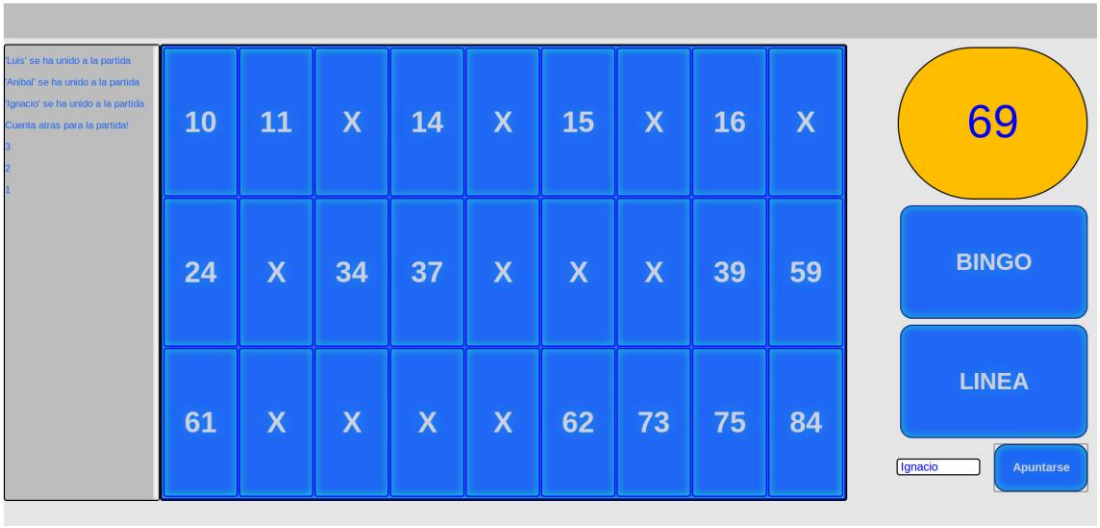


Imagen 75. Pantalla de juego

En la parte superior derecha salen los números del “bombo” y en la parte izquierda aparece un “log” que aporta toda la información del transcurso de la partida.

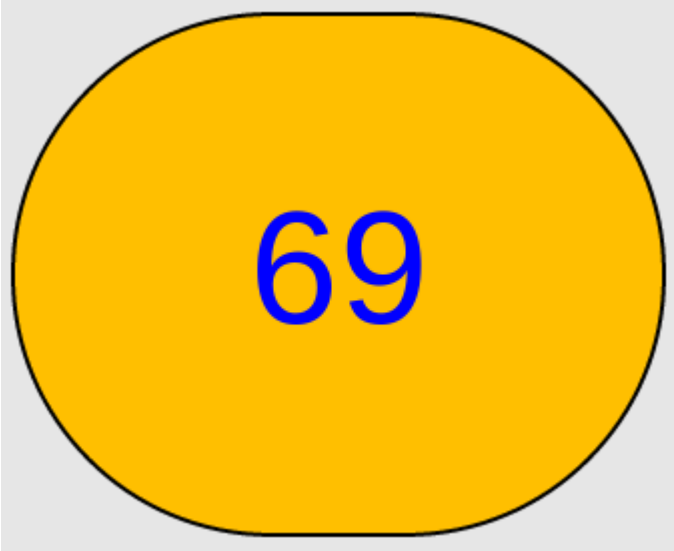


Imagen 77. Bombo del juego

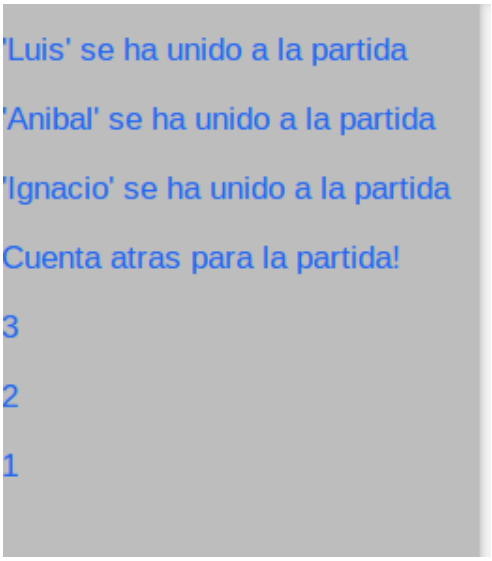


Imagen 76. Log

Con la partida en curso el jugador puede hacer click en los botones “BINGO” y “LINEA” cuando estime conveniente para ganar la partida.

Imagen 78. Botones Bingo y Linea

- Reglas:

La partida estará formada por un número mínimo de usuarios configurado por el administrador.

Los números aleatorios del bombo pertenecen a un intervalo del 1 al 99.

Sólo se podrá cantar una línea válida y habrá un único ganador de línea, cuando todos los números de cualquier fila del cartón salgan en el bombo.

Habrá un ganador único cuando todos los números del cartón salgan en el bombo, momento en el cual podrá cantar bingo.

CAPÍTULO 8

CONCLUSIONES

8.1 – Conclusiones personales:

Las conclusiones positivas que puedo aportar tras la consecución del proyecto son las siguientes:

- Es un lenguaje con una gran variedad de frameworks y herramientas para desarrollar con mayor facilidad toda clase de sistemas. Las herramientas son las justas sin la posibilidad de caer en una confusión ante una oferta excesiva de todo tipo de herramientas como ocurre en otras comunidades.

- La documentación es corta y precisa y en su amplia mayoría se encuentra en inglés.

- El lenguaje es libre lo que nos permite desarrollar sin tener que realizar gasto económico alguno y esto posibilita el acceso a un mayor número de desarrolladores para unirse a la comunidad para evolucionar el lenguaje.

- Es un lenguaje que dado a la orientación que está cobrando el mundo de la tecnología a aplicaciones distribuidas, se convierte en un lenguaje muy potente y útil.

Las conclusiones negativas que pudieron observarse son:

- El aspecto abstracto de muchos conceptos como son los procesos, que alargaron el tiempo de aprendizaje.

8.2 – Consecución de objetivos:

Se puede afirmar que se ha completado el proyecto software, desarrollando una aplicación con tecnología servidor Erlang y tecnología cliente HTML + JS.

Dicha aplicación cumple todos los objetivos para los que fue diseñada, la robustez del sistema, la seguridad, un procesado veloz con muchas conexiones, una fácil escalabilidad y mantenimiento, etc…

El proyecto se centraba en la tecnología servidor donde se pretende y consigue ofrecer un sistema online altamente concurrente utilizando tecnología Websocket.

A lo largo de la realización del proyecto se tuvo muy presente la importancia de la modularidad y la escalabilidad para conseguir que sea fácil de modificar si se quieren añadir ampliaciones y que pueda dar soporte a un mayor número de usuarios en caso de éxito comercial.

8.3 – Posibles ampliaciones:

Se pensó acerca de algunas ampliaciones que se podrían aplicar al proyecto. Estas ideas fueron apareciendo a medida que se avanzaba en el desarrollo y al finalizar este. Son las siguientes:

- Dotar al sistema de funcionalidad para manejar partidas apostando dinero y así premiar también las líneas.

- Agregar persistencia al proyecto migrando los datos de la tabla ETS a una base de datos.

- Diseñar una pantalla de bienvenida donde los usuarios puedan elegir qué tipo de partidas desean jugar, filtrándolas por nivel económico.

- Ofrecer una opción multilenguaje para internacionalizar la aplicación.

BIBLIOGRAFÍA

Bibliografía:

Libros:

"Erlang and OTP in action", Martin Logan, Eric Merritt, Ed.Manning

"Building Web Applications with Erlang", Zachary Kessin, Ed O´Reilly

"Introducing Erlang", Simon St.Laurent, Ed O´Reilly

"Programming Erlagn", Joe Amstrong, The Pragmatic Programmers

"Erlang/OTP System Documentation", Ericcson AB.

Páginas Web:

http://learnyousomeerlang.com/

https://www.erlang-solutions.com/

http://aprendiendo-erlang.blogspot.com.es/

http://www.erlang.org/

http://ninenines.eu/

http://json.org/

http://www.w3schools.com/jsref/