



Bienvenidos a la segunda clase de Erlang.

Ejercicio: Programar un servicio de Broadcast

```
-export([iniciar/0, finalizar/1]).  
-export([broadcast/2, registrar/1]).  
-export([loopBroadcast/1]).
```

Un servicio de Broadcast lo podemos pensar como un objeto concurrente que acepta dos funciones:

- **Registrar()** función que registra el proceso que invoca la función
- **Broadcast(Msg)** que envía un mensaje a los procesos que se han registrado.

Un servicio de Broadcast lo podemos pensar como un objeto concurrente que acepta dos funciones:

- Registrar() función que registra el proceso que invoca la función
- Broadcast(Msg) que envía un mensaje a los procesos que se han registrado.

Ayuda: Utilizar la función `lists:foreach(fun (Pid) -> Pid ! Msg end , PidsList)`,



Clase 2

- Receive (Timeouts)
- Registración de Procesos
- Revisitamos Modelo Cliente/Servidor
- Breve mención al Scheduler de Procesos
- Manejo y propagación de Errores

En esta clase entraremos en algunos detalles más sobre el lenguaje.

Reforzaremos el concepto de buzón de mensajes, y veremos una técnica para darle prioridad a ciertos mensajes.

Veremos cómo asignarles nombres a ciertos procesos, facilitando la implementación de librerías, etc.

Revisitaremos el modelo cliente/servidor desde el punto de vista de Erlang.

Revisaremos el concepto de scheduler y veremos como reducir la prioridad de procesos en Erlang.

Finalmente veremos cómo hacer un manejo buen manejo de errores en Erlang.



Receive

```
receive  
  Patrón1 -> Cuerpo1  
  Patrón2 -> Cuerpo2  
  ...  
  PatrónN -> CuerpoN  
end
```

La primitiva de concurrencia **receive** nos permite buscar mensajes en el buzón.

El proceso de Pattern Matching se basa en buscar en el orden de llegada de los mensajes, el primer mensaje que pueda ser matcheado con alguno de los patrones.

Receive nos permite hacer pattern matching con los mensajes que estén en el buzón. El proceso de pattern matching es entonces en orden. Esto es, en el orden en que llegaron los mensajes, se comparan con los patrones en el orden en el que son escritos y en la primer coincidencia, se instancian las variables necesarias y se ejecuta el cuerpo correspondiente.

Debido a que dependemos en qué orden vinieron los mensajes, el orden de las guardas **no nos garantiza ninguna prioridad**.

Receive con Timeout

```
receive
  Patrón1 -> Cuerpo1
  Patrón2 -> Cuerpo2
  ...
  PatrónN -> CuerpoN
after
  ExprTO -> CuerpoT
end
```

Al agregar la cláusula **after** podemos estipular un tiempo de espera, que puede ser calculado en tiempo de ejecución.

Hay dos tiempos (átomos) que son útiles:

- **0** (número cero)
- **infinity**

La expresión `ExprTO` es una expresión que evalúa a un entero interpretado en milisegundos (la precisión es dependiente de la implementación de Erlang y el sistema op).

- Cuando se ingresa a un receive, se inicia un timer (pero solo si una sección after está presente en la expresión). Con un valor cero, el tiempo de espera se produce inmediatamente si no hay ningún mensaje coincidente en la cola de mensajes. El átomo infinito hará que el proceso espere indefinidamente un mensaje coincidente. Esto es lo mismo que no usar un tiempo de espera. Puede ser útil para valores de tiempo de espera que se calculen en tiempo de ejecución.
- Toma el primer mensaje del buzón e intenta compararlo con Pattern1, Pattern2, etc. Si el matching tiene éxito, el mensaje se elimina del buzón y se evalúan las expresiones que siguen al patrón.
- Si ninguno de los patrones en la declaración de recepción coincide con el primer mensaje en el buzón, entonces el primer mensaje se elimina del buzón y se coloca en una "cola de guardado". A continuación, se intenta con el segundo mensaje del buzón. Este procedimiento se repite hasta que se encuentra un mensaje que matchee o hasta que se han examinado todos los mensajes del buzón.
- Si ninguno de los mensajes en el buzón matchea, el proceso se suspende

- y se reprogramará para ejecutarse la próxima vez que se coloque un nuevo mensaje en el buzón. Tenga en cuenta que cuando llega un mensaje nuevo, los mensajes en la cola de guardado no se vuelven a rematchear; solo el nuevo mensaje es matcheado.
- Tan pronto como se ha encontrado una coincidencia con un mensaje, todos los mensajes que se han puesto en la cola de guardado se vuelven a ingresar en el buzón en el orden en que llegaron al proceso. Si se configuró un temporizador, se borra.
- Si se cumple el tiempo de espera cuando se está esperando un mensaje, entonces se evalúa la expresión ExpressionTimeout y se vuelven a colocar los mensajes guardados en el buzón en el orden en que llegaron al proceso.

Ejercicios utilizando `after`

Implementar las siguientes funciones:

- `sleep/1` bloquea por cierto tiempo
- `empty_mailbox/0` vacía el buzón de un proceso
- `priority_process/0` dados dos mensajes, `msg1` y `msg2`, hace que `priority msg1` tenga más prioridad que `msg2`.



Registración de Procesos

Toda la comunicación entre procesos en Erlang es a través del identificador del proceso, pero no siempre es práctico (ni deseable). Para esto se le puede asignar un nombre (átomo) a un identificador de procesos.

- Mejora la legibilidad y comprensión del código
- Permite la ubicación y el descubrimiento de procesos
- Flexibilidad y modularidad

Puede cambiar el proceso que actualmente está siendo el que espera a los clientes. Tiene sentido tener una entidad un poco más abstracta y tener nombres.

- Mejora la legibilidad y comprensión del código
- Permite la ubicación y el descubrimiento de procesos. Al registrar procesos con nombres, otros procesos pueden buscarlos y descubrirlos fácilmente en el sistema Erlang.
- Flexibilidad y modularidad: La registración de procesos permite una mayor flexibilidad y modularidad en el diseño del sistema. Los nombres registrados pueden servir como puntos de entrada para diferentes funcionalidades o módulos del sistema, lo que facilita la escalabilidad y el mantenimiento del código.



BIFs Registración de Procesos

- **register (Name, Pid)** : registra al identificador de proceso **Pid** con el nombre **Name**
- **unregister (Name)** : borra el nombre **Name** del registro
- **whereis (Name)** : retorna el identificador de proceso asociado a **Name**
- **registered ()** : retorna una lista de los procesos registrados en el sistema

Notar que es `whereis` y no `whois`. El PID nos da información sobre DÓNDE ESTÁ EL PROCESO!

BIFs = Built-In Functions

Ejercicio: Programar un servicio de Broadcaster

Registrar el proceso como broadcaster y propagar el cambio:

```
-export([iniciar/0, finalizar/0]).  
-export([broadcast/1, registrar/0]).
```

Donde `iniciar/0` ya no dará el PId del proceso encargado de administrar los mensajes, sino que registrará ese proceso con nombre `broadcaster`. Broadcast lo que hace simplemente es estar a la espera de mensajes

Modelo Cliente/Servidor

Revisitamos el modelo cliente/servidor, y para esto nos viene bien registrar nombres!!

En particular, para el modelo cliente-servidor estas características son muy útiles y se ven reflejadas de la siguiente manera:

- Facilita la comunicación cliente-servidor: Los clientes pueden utilizar los nombres registrados para enviar mensajes directamente a los servidores sin tener que conocer los Pid específicos de cada servidor.
- Escalabilidad y reconfiguración: En entornos distribuidos o de escala empresarial, donde los servidores pueden estar distribuidos en diferentes nodos o pueden ser dinámicamente agregados o eliminados, la registración de nombres permite una mayor escalabilidad y flexibilidad. Los clientes no necesitan conocer la ubicación exacta o los Pid de los servidores, ya que pueden utilizar los nombres registrados para localizarlos automáticamente, independientemente de su ubicación o estado actual.
- Resiliencia y tolerancia a fallos: La registración de nombres de identificadores de procesos también es útil para lograr resiliencia y tolerancia a fallos en sistemas Erlang. Si un servidor falla o se reinicia, se puede registrar un nuevo proceso con el mismo nombre una vez que esté nuevamente en funcionamiento. Los clientes no necesitan preocuparse por los detalles internos o las interrupciones del servidor, ya que pueden seguir utilizando el nombre registrado para comunicarse con él una vez que se recupera.
- Modularidad y mantenibilidad: La registración de nombres promueve la modularidad y la mantenibilidad del código. Los servidores pueden estar

- implementados como módulos separados y los clientes pueden interactuar con ellos a través de los nombres registrados, lo que permite una mejor organización del código y facilita el mantenimiento y la extensibilidad del sistema.

Componentes Modelo Cliente/Servidor

Servidor

Proceso que otorga el servicio

Protocolo

Comunicación cliente-servidor

Librería

Batería de funciones que dan acceso a los servicios del servidor.

Definamos mejor el modelo cliente servidor.

Lo podemos descomponer en tres:

- + Servidor: **un** proceso que ofrece algún servicio (esto ya lo teníamos)
- + Protocolo: la forma y procedimiento utilizado para comunicarnos con el servidor
- + Librería de acceso: las funciones que nos permiten interactuar con el servidor (funciones que implementan el protocolo).

El servidor y el protocolo es algo que ya vimos en C. Veamos mejor entonces lo que sería la librería de acceso.



Librería de Acceso

Las funciones de acceso presentan una capa de abstracción que oculta el protocolo de comunicación con el servidor.

De esta manera podemos desconectar lo más posible al cliente:

- Oculta las componentes internas del servidor: Donde está, quién es, pueden ser incluso múltiples servidores, etc.
- Nos permite además cambiar la implementación libremente: Básicamente como una librería.

La idea es abstraer lo más posible al cliente (o usuario de nuestro servicio) de los protocolos internos de comunicación con el Servidor.

Ejercicio: Analizar el Broadcaster con el Modelo Cliente-Servidor

Preguntas:

1. ¿Quién es el Servidor?
2. ¿Cuál es el protocolo de comunicación? ¿El cliente se entera si una operación no pudo realizarse?
3. ¿Cuál sería la Librería de Acceso del Servidor?

Repasemos el Broadcaster entre todos.

Recordar implementar un Broadcaster confiable, es decir, con acuse de recepción por parte del servidor.

Servidor de Broadcasting

Servidor

Proceso
broadcaster

Protocolo

El proceso Servidor recibirá por mensaje qué operación un cliente requiere y responderá un mensaje de éxito o de error.

Librería

Funciones **iniciar/0**
finalizar/0
enviar/1
subscribir/1

Revisar broadcasting utilizando lo que sabemos. Registración de procesos, y modelo cliente-servidor.



Scheduler de Erlang

La distribución de la unidad de cómputo en Erlang es responsabilidad de la máquina virtual (BEAM).

- Scheduling **Justo**: todo proceso que pueda ejecutarse, se ejecutará.
- Ningún proceso se apropiará indefinidamente de la unidad de cómputo.

El scheduler de Erlang está diseñado específicamente para la ejecución de procesos Erlang y se enfoca en la planificación eficiente y justa de estos procesos, mientras que **el scheduler del sistema operativo Linux tiene una función más general** de administración de recursos y planificación de todos los procesos en el sistema operativo.

La distribución de la unidad de cómputo en Erlang es responsabilidad de la máquina virtual. Recordemos que Erlang ejecuta sobre una máquina virtual y **no** sobre el sistema operativo directamente. Esto nos trae un montón de beneficios que son los que ya vimos.

Pero el se le pide que la repartición de tareas sea:

- + Justa: todo proceso que pueda ser ejecutado será ejecutado realmente (Fair Scheduling)
- + Ningún proceso se puede apropiar del cpu indefinidamente

De hecho hay una unidad llamada `time slice` (porción de tiempo) que suele ser de `500` reducciones (llamado de funciones).

1. **Planificación eficiente:** El scheduler de Erlang implementa un planificador de hilos propio que administra la asignación de tiempo de CPU a los procesos y garantiza una planificación equitativa y eficiente. El planificador utiliza técnicas como el encolamiento múltiple y la conmutación voluntaria para minimizar el tiempo de espera y maximizar la utilización de la CPU.
2. **Soporte para concurrencia y paralelismo:** Erlang se destaca por su capacidad para manejar una gran cantidad de procesos concurrentes de manera eficiente. El scheduler de Erlang permite la ejecución concurrente y paralela de procesos al asignarlos a hilos subyacentes. Esto facilita la construcción de sistemas escalables y tolerantes a fallos en Erlang.
3. **Tiempo de respuesta constante:** El scheduler de Erlang se esfuerza por proporcionar un tiempo de respuesta constante a los procesos, lo que significa que incluso en sistemas altamente cargados, los procesos críticos y

1. prioritarios pueden recibir una porción justa y predecible del tiempo de CPU. Esto es esencial para aplicaciones en tiempo real y de alta disponibilidad.
2. **Aprovechamiento de múltiples núcleos:** El scheduler de Erlang está diseñado para aprovechar eficientemente los múltiples núcleos de las máquinas modernas. Puede asignar procesos a diferentes núcleos de manera inteligente para lograr un mayor rendimiento y una mejor utilización de los recursos del sistema.
3. **Tolerancia a fallos:** El scheduler de Erlang es un componente crítico en la construcción de sistemas tolerantes a fallos en Erlang. Permite que los procesos se supervisen y se tomen medidas en caso de que ocurra un fallo. Esto incluye la capacidad de reiniciar procesos fallidos, redistribuir la carga entre los procesos activos y recuperarse de situaciones anómalas.



Scheduler de Erlang

Dispone de un sistema de prioridades donde aquellos procesos con mayor prioridad, ejecutan más seguido.

La prioridad se indica con: **`process_flag(priority, Level)`**

- `normal` (default)
- `low`
- `high`
- `max`

Los procesos se ejecutan con prioridad `normal` por defecto. Ejemplo de uso:
`process_flag(priority, high).`

¿Quién se encargaba de asignarles tiempo de ejecución antes?

Manejo de Errores

Veremos entonces unos mecanismos para el manejo de errores en Erlang. Como se espera, al ser un lenguaje con tipado dinámico tendremos errores de runtime (como en C).



Excepciones

Las excepciones se utilizan para indicar detener la ejecución de un proceso e indicar un estado de error.

- **Errores en tiempo de ejecución:** Cuando crashea un proceso. También se puede emular un error en tiempo de ejecución llamando a **error (Reason)**. Los errores en tiempo de ejecución son excepciones de la clase **error**.
- **Errores generados:** Cuando el propio código llama a **exit/1** o **throw/1**. Los errores generados son excepciones de las clases **exit** o **throw**.

El mecanismo consiste de dos partes, una que dispara la excepción y otra que la captura.

Excepciones comunes en Erlang

badmatch

Error en un matching

badarg

Error en el tipo de argumento

case_clause

Ninguna cláusula de un case es válida

if_clause

Ninguna cláusula de un if es válida

undef

invocar a una función que no existe

badarith

error en una operación aritmética

- + Error badmatch : ``1 = 3`` lanza la excepción `{EXIT, PId, badmatch}`
- + Error badarg: Una BIF se llamó con un tipo incorrecto,
- + Error case_clause: `M = 3, case M of`
 - `1 -> true;`
 - `2 -> false`
 - `end.`
- + Error if_clause: lo mismo que case, ninguna guarda se cumple
- + Error undef: llamar a una función que no existe, ``foo(1)`` mientras que foo no se declaró en ninguna lado. O no se exportó!
- + Error badarith: mala operación aritmética, como dividir por 0.



Captura de Excepciones

Para capturar una excepción lo hacemos de la siguiente forma: **catch Expr.**

- **catch Expr** devuelve el valor de **Expr** a menos que ocurra una excepción durante la evaluación. En dicho caso, se captura la excepción.
- En el caso que se dispare una excepción será una tupla con información de la excepción **{ 'EXIT' , {Reason, Stack} }**

La captura de excepciones cobrará sentido cuando vean cómo disparar una excepción.

El Reason depende del tipo de error que ocurrió, y Stack es el stack de llamadas a funciones recientes



Lanzar Excepciones

Al lanzar una excepción el proceso corta abruptamente su ejecución y envía una excepción hacia atrás hasta encontrar un **catch** que la maneje.

Lo logramos con la BIF **throw/1**.

La idea de throw es simplemente lanzar una excepción, y enviarla “hacia atrás” hasta encontrar un catch que la reciba

Ejemplo división por cero

Escribir una función que capture la excepción al realizar la división por cero, e imprima por pantalla un mensaje indicando que la excepción fue capturada.



Terminación de Procesos

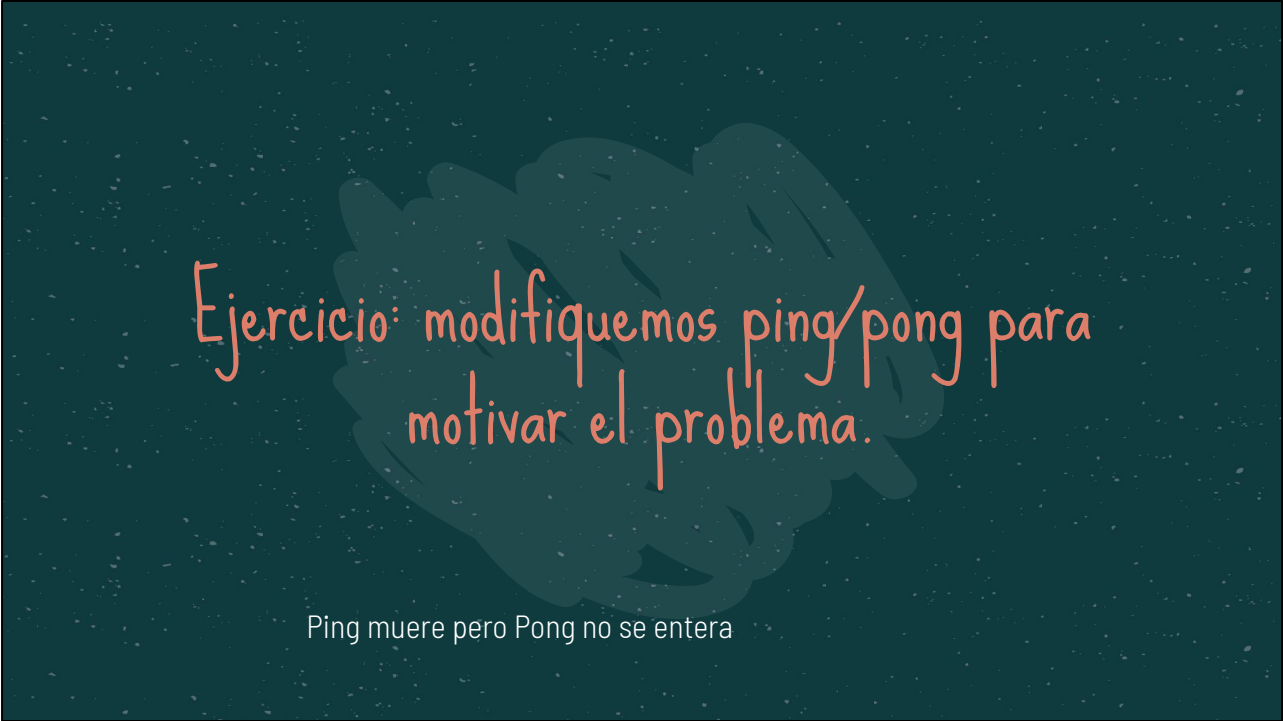
Los procesos en Erlang terminan correctamente si:

- Invocan a **exit(normal)**, o
- No tienen más nada que ejecutar

Antes de ver los errores, veamos como termina correctamente un proceso en Erlang.
Todo proceso que tiene un error en ejecución (runtime error):

- + División por cero
- + Error en el pattern matching
- + Invocar a una función que no existe

etc. Termina de forma **abnormal** invocando a la función `exit(Reason)`, siendo Reason distinta al átomo ``normal``



Ejercicio: modifiquemos ping/pong para motivar el problema.

Ping muere pero Pong no se entera

Utilizar el comando `regs()`. en Eshell para visualizar todos los procesos.



Linkeo de Procesos

- Los procesos pueden indicar cuando terminan de forma anormal mediante un sistema de links. Lo creamos con la primitiva **link (PidOtro)**.
- **link/1** crea una conexión **bidireccional** entre el proceso que la invoca y el que es pasado como argumento.
- Cuando un proceso termina, le envía una señal a todos los procesos a los que estaba conectado con la razón por la que terminó. Por defecto, el programa que recibe la señal **normal** la ignora. En el caso que sea **abnormal** u otra razón, puede pasar lo siguiente:
 - Enviar todos los mensajes del buzón de un proceso al otro
 - Morir
 - Propagar el comportamiento a los procesos linkeados por el receptor.

La idea es básicamente que Erlang al crear procesos para todo, estos pueden fallar y otros procesos no estar ni enterados que hubo un error en el sistema.

La idea de generar un `link` entre dos procesos es para evitar dicho problema, y **cuando un proceso falla, el otro también.**

`link` crea una conexión bidireccional entre el proceso que la invoca y el que es pasado como argumento.

Cuando un proceso termina, le envía una señal a todos los procesos a los que estaba conectado con la razón por la que terminó.

Por defecto, el programa que recibe la señal `normal` la ignora.

En el caso que sea `abnormal` u otra razón, puede pasar lo siguiente:

- + Enviar todos los mensajes del buzón de un proceso al otro
- + Morir
- + Propagar el comportamiento a los procesos linkeados por el receptor.

Ejercicio

Resolver el problema de que si muere ping o pong pero el otro no se enteró.
Solución: pong debe morir también

Resolver el problema de que muera ping o pong pero el otro no se enteró.
La solución es que pong muere también



Ante una Terminación Anormal

Se puede:

- **Detectar la señal:** El proceso que recibe la señal la detecta como un mensaje más

O bien,

- **Matar al proceso y propagar la señal:** Se mata el proceso que recibe la señal y se propaga la señal a los procesos que están linkeados a este último.

Cuando se produce una terminación con una razón distinta a `normal` se puede capturar la señal.

De esta manera uno puede conectar los procesos, y si uno falla por algún motivo, el resto se puede enterar y:

- + Tratar de solucionar el error
- + Abandonar el barco silenciosamente
- + Avisarle al resto y morir juntos

Ejercicio

BIF útil: `spawn_link`

Para poder detectar la señal hay que indicárselo a Erlang con:

`process_flag(trap_exit, true)`

con `spawn_link` creamos un proceso y generamos la conexión al mismo tiempo.

con `process_flag(trap_exit, true)` activamos la captura de la señal de muerte.

Y viene de la siguiente forma `{'EXIT', From, Reason}`

Entonces ping muere, pong se entera y decide terminar elegantemente.



Qué diferencia hay entonces entre el uso de catch y linkeo de programas?

La diferencia radica en el uso. Las excepciones y catch están pensadas para que las use un proceso, y el linkeo para propagar errores a través de la red de procesos. Ambas funciones permiten implementar software robusto, un proceso puede recuperarse de un error, y un sistema puede recuperarse en el caso que un proceso fallé.

Por ejemplo, podemos tener un proceso llevando cuenta del sistema, y mirando que el proceso que brinda el servicio esté vivo. Si el servidor falla, el monitor puede revivirlo.