# Programación Declarativa Avanzada

# Erlang 2

Juan Rodríguez Hortalá y el equipo de erlang.org

# Programación tolerante a fallos

Ya hemos visto como se implementa el actor model en Erlang

Cada proceso es un actor
Los actores se envían mensajes, que son términos Erlang
Los procesos tiene un buzón FIFO: los procesos pueden ejecutar una instrucción `receive` para procesar los mensajes que encajen con algún patrón

Ahora veremos los mecanismos que ofrece Erlang para la programación tolerante a fallos facilita construir sistemas de alta disponibilidad (servidores que siempre deberían estar operativos)

Enfoque "**let it crash** and let someone else deal with" y "crash early"

. si algo va mal deja que el proceso Erlang involucrado termine cuando antes y crea un proceso nuevo que se encargue del problema
. de esta manera se aísla el error y se asegura que el sistema sigue operativo sin pausas
. se debe evitar la programación defensiva

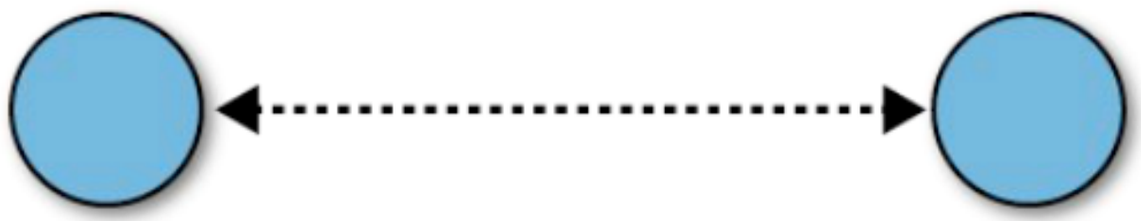
# Enlazando procesos Erlang

## Enlace bidireccional la BIF `link/1`

`link(Pid)` crea un enlace bidireccional entre el proceso que llama a la función y el proceso especificado por `Pid`
`spawn_link(Fun), spawn_link(Module, Function, Args)` equivalente a llamar a `spawn` y luego a `link` (pero las dos llamadas se ejecutan atómicamente)

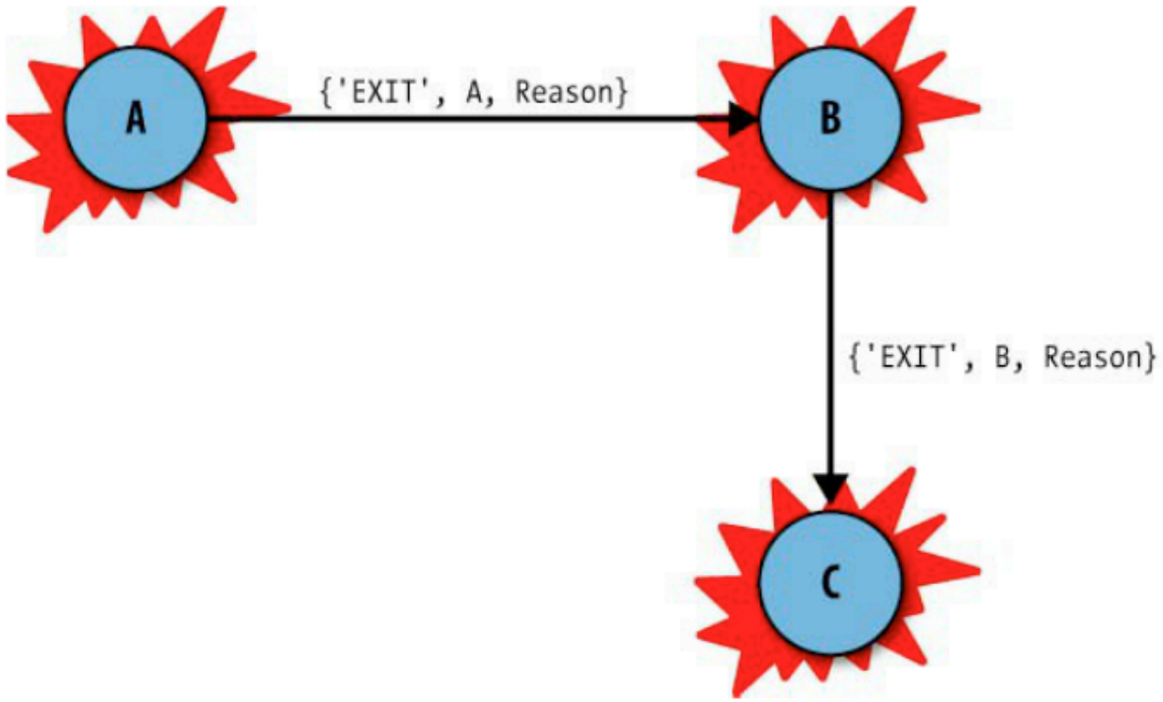
Enlace bidireccional porque que el proceso A llame a `link` sobre el proceso B es que equivalente a que el proceso B llame a `link` sobre el proceso A.

Diagrama que representa el enlace



# Enlazando procesos Erlang

Si un proceso termina anormalmente (fallo de matching, excepción, ...) entonces se enviará una exit signal a todos los otros procesos a los que está enlazado, y cada uno de ellos a su vez también terminará enviando otra exit signal a los demás procesos a los que esté enlazado.




- La misma razón se propaga en la cadena de procesos enlazados

- Si varios procesos dependen mutuamente unos de otros para funcionar correctamente entonces es una buena práctica enlazarlos para asegurar que cuando uno aborte entonces los demás también lo hagan no pueden trabajar bien si no están activos todos los demás

Demo **`add_one.erl`** parte uno

# Capturando señales

Con lo visto hasta ahora lo único que puede hacer un proceso para reaccionar a la caída de otro procesos al que está enlazado es terminarse a sí mismo con eso no podemos hacer que el sistema se recupere

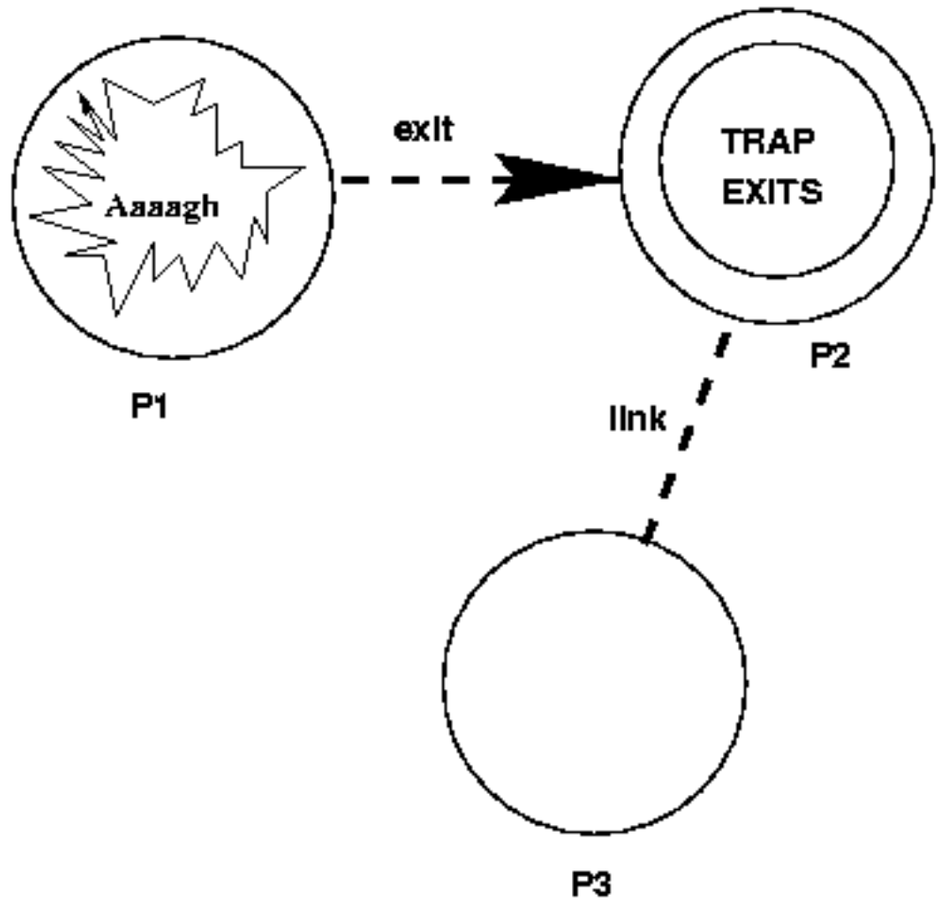
`process_flag(trap_exit, true)` el proceso actual convertirá las exit signal en exit messages de la forma `{'EXIT', Pid, Reason}` donde `Pid` es el identificador del procesos que se está terminando, y `Reason` es un término Erlang explicando la razón de que termine. Se dice que el proceso está ***trapping exits***

Estos mensajes se pueden procesar en un `receive` por el proceso enlazado, que en vez de morir puede hacer las acciones que necesite para recuperar el sistema: por ejemplo volver a poner en marcha un servicio que se acaba de caer creando un nuevo proceso que lo ejecute

Demo `add_one.erl` trapping exits

# Processes can trap exit signals

In the following diagram P1 is linked to P2 and P2 is linked to P3. An error occurs in P1 - the error propagates to P2. P2 traps the error and the error is **not** propagated to P3.




P2 has the following code:

```
process_flag(trap_exit, true)
receive
     {'EXIT', P1, Why} ->
   ... exit signals ...
     {P3, Msg} ->
   ... normal messages ...
end
```

# Capturando señales

La señal `kill` lanzada con `exit(Pid, kill)` nunca se puede capturar y siempre resulta en que el proceso Pid se termina con `killed` como razón de esta manera no matará a los procesos a los que esté enlazados porque a estos se les mandará `killed` en vez de `kill`

Con captura de señales podemos ver la importancia de que `spawn_link` sea atómico: permite obtener el motivo del error

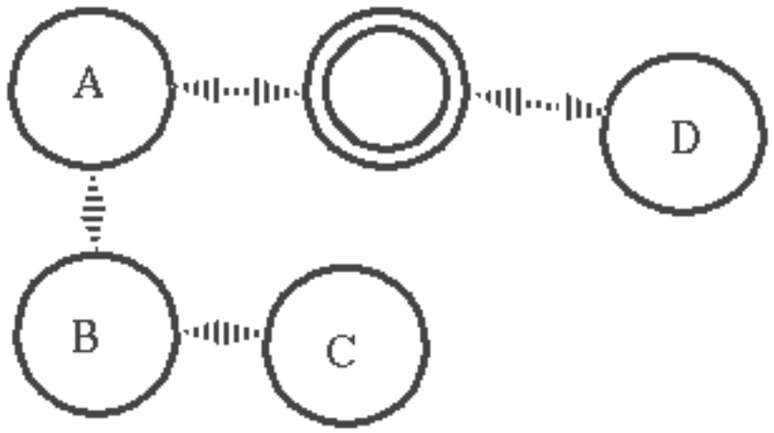
```
> self() .
> process_flag(trap_exit, true), spawn_link(fun() -> 1/0 end) .
> self() .
> flush() .

> self() .
> process_flag(trap_exit, true), Pid = spawn(fun() -> 1/0 end) .
> self() .
> link(Pid) .
> flush() .
```

Mediante unlink podemos deshacer un enlace entre dos procesos suele ser recomendable vaciar el buzón de los posibles señales capturadas anteriormente

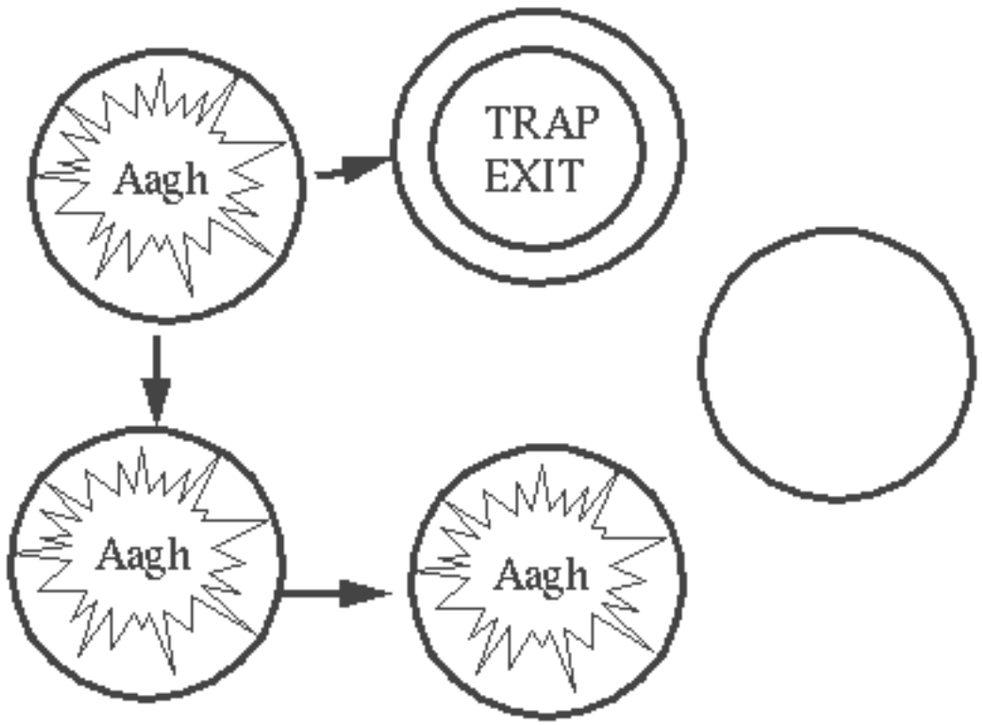
```
unlink(Pid),
    receive
        {'EXIT', Pid, _} ->
            true
    after 0 ->
            true
```

# Complex Exit signal Propagation

Suppose we have the following set of processes and links:




The process marked with a *double ring* is an error trapping process.




If an error occurs in any of A, B, or C then *All* of these process will die (through propagation of errors). Process D will be unaffected.

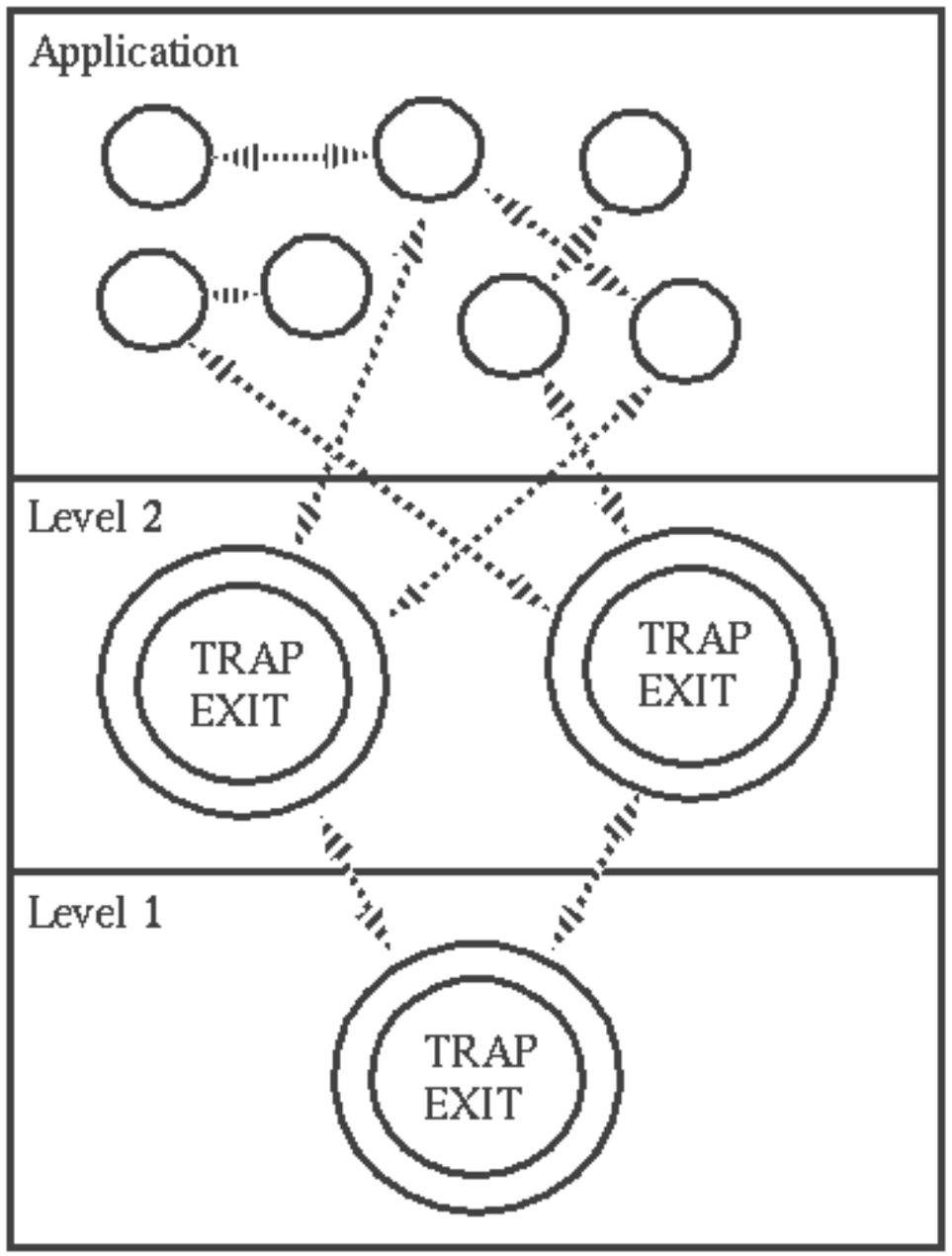
# Exit Signal Propagation Semantics

- When a process terminates it sends an exit signal, either normal or non-normal, to the processes in its link set.
- A process which is not trapping exit signals (a normal process) dies if it receives a non-normal exit signal. When it dies it sends a non-normal exit signal to the processes in its link set.
- A process which is trapping exit signals converts all incoming exit signals to conventional messages which it can receive in a receive statement.
- Errors in BIFs or pattern matching errors send automatic exit signals to the link set of the process where the error occured.

# Robust Systems can be made by Layering

By building a system in layers we can make a robust system.

- Level1 traps and corrects errors occuring in Level2.
- Level2 traps and corrects errors ocuring in the application level.




In a well designed system we can arrange that application programers will not have to write any error handling code since all error handling is isolated to deper levels in the system.

# Primitives For Exit Signal Handling

- **link(Pid)** - Set a bi-directional link between the current process and the process **Pid**
- **process_flag(trap_exit, true)** - Set the current process to convert exit signals to exit messages, these messages can then be received in a normal receive statement.
- **exit(Reason)** - Terminates the process and generates an exit signal where the process termination information is **Reason.**

What really happens is as follows: Each process has an associated mailbox - **Pid!Msg** sends the message **Msg** to the mailbox associated with the process **Pid**.

The **receive .. end** construct attempts to remove messages from the mailbox of the current process. Exit signals which arrive at a process either cause the process to crash (if the process is not trapping exit signals) or are treated as normal messages and placed in the process mailbox (if the process is trapping exit signals). Exit signals are sent implicitly (as a result of evaluating a BIF with incorrect arguments) or explicitly (using **exit(Pid, Reason)**, or **exit (Reason)** ).

If **Reason** is the atom **normal** - the receiving process ignores the signal (if it is not trapping exits). When a process terminates without an error it sends normal exit signals to all linked processes. *Don't say you didn't ask!*

# A Robust Server

The following server *assumes* that a client process will send an **alloc** message to allocate a resource and then send a **release** message to deallocate the resource.

This is the top loop of an allocator with no error recovery.

- **Free** is a list of unreserved resources.
- **Allocated** is a list of pairs **{Resource, Pid}** showing which resource has been allocated to which process.

```
top(Free, Allocated) ->
    receive
   {Pid, alloc} ->
       top_alloc(Free, Allocated, Pid);
   {Pid ,{release, Resource}} ->
       Allocated1 = delete({Resource,Pid},Allocated),
       top([Resource|Free], Allocated1)
    end.

top_alloc([], Allocated, Pid) ->
    Pid ! no,
    top([], Allocated);

top_alloc([Resource|Free], Allocated, Pid) ->
    Pid ! {yes, Resource},
    top(Free, [{Resource,Pid}|Allocated]).
```

This is unreliable - *What happens if the client crashes before it sends the release message?*

# Allocator with Error Recovery

The following is a *reliable* server. If a client craches *after* it has allocated a resource and *before* it has released the resource, then the server will automatically release the resource (`process_flag(trap_exit, true)` is called before starting the loop `top_recover/2`)

```
top_recover(Free, Allocated) ->
    receive
    {Pid , alloc} ->
        top_recover_alloc(Free, Allocated, Pid);
    {Pid, {release, Resource}} ->
        unlink(Pid),
        Allocated1 = delete({Resource, Pid}, Allocated),
        top_recover([Resource|Free], Allocated1);
    {'EXIT', Pid, Reason}->
        %% No need to unlink.
        Resource = lookup(Pid, Allocated),
        Allocated1 = delete({Resource, Pid}, Allocated),
        top_recover([Resource|Free], Allocated1)
    end.

top_recover_alloc([], Allocated, Pid) ->
    Pid ! no,
    top_recover([], Allocated);

top_recover_alloc([Resource|Free], Allocated, Pid) ->
    Pid ! {yes, Resource},
    link(Pid),
    top_recover(Free, [{Resource,Pid}|Allocated]).
```

The server is linked to the client during the time interval when the resource is allocted. If an exit message comes from the client during this time the resource is released.

## Allocator Utilities

```
delete(H, [H|T]) ->
    T;
delete(X, [H|T]) ->
    [H|delete(X, T)].

lookup(Pid, [{Resource,Pid}|_]) ->
    Resource;
lookup(Pid, [_|Allocated]) ->
    lookup(Pid, Allocated).
```

Not done -- multiple allocation to same process. i.e. before doing the **unlink(Pid)** we should check to see that the process has not allocated more than one device.

# Enlaces unidireccionales

Enlace unidireccional la BIF `monitor/2` es similar a `link/1` pero unidireccional

`monitor(process, Pid), monitor(process, RegName)` crea un enlace unidireccional entre el proceso que llama a la función (proceso monitor) y el proceso especificado (proceso monitorizado), y devuelve un valor de tipo referencia que identifica a la monitorización

- si el proceso monitorizado muere entonces un mensaje `{'DOWN', MonitorRef, process, Object, Info}` será enviado al proceso monitor donde
  . `MonitorRef` es la misma referencia que se devolvió en la llamada a `monitor/2`
  . `Object` será el `Pid` del proceso monitorizado o `{RegName, Node}`, según lo que se pasara en la llamada a `monitor/2` (RegName es un nombre registrado con register/2, Node indica información de computación distribuida entre varias máquinas que por ahora ignoraremos)
  . `Info` será la exit reason del proceso monitorizado, o bien `noproc` (el proceso no existe) o `noconnection` (no se puede conectar con el proceso, para computación distribuida)
- el enlace es unidireccional porque el proceso monitorizado no recibe ningún mensaje si el proceso monitor se muere
- cuando se pasa un `RegName` entonces se monitorizará siempre el pid que estaba asociado a ese nombre en el momento de la llamada a monitor/2, sin importar que más adelante se deshaga esa asociación mediante `unregister/1`

# Enlaces unidireccionales

- si intentamos monitorizar un proceso que no existe entonces no se produce un error de ejecución sino que un mensaje `{'DOWN', MonitorRef, process, Object, noproc}` se envía inmediatamente al proceso monitor
- además se pueden crear varios monitores para el mismo proceso, los cuales mandarán cada uno si propio mensaje `DOWN` cuando el proceso monitorizado muera

```
> Pid = spawn(mal, no_existe, []) .
> Ref = monitor(process, Pid) . % el proceso consola es el monitor
> receive {'DOWN', Ref, process, Pid, Reason} -> {Pid, Reason} end .
```

pero tampoco vale cualquier cosa

```
> Ref2 = monitor(process, 0) .
```

# Enlaces unidireccionales

`spawn_monitor(Fun, Pid), spawn_monitor(Module, Function, Args)` similar a las variantes de `spawn_link`

`demonitor(MonitorRef)` el proceso que llama a la función deja de realizar la monitorización correspodiente a `MonitorRef`, que es una referencia obtenida en una llamada previa a `monitor/2`. Si se usa `demonitor(MonitorRef, [flush])` entonces adicionalmente se eliminan del mailbox del proceso invocador todos los mensajes correspondientes a la monitorización especificada

# Cambio de código en caliente

Es común encontrar un error en un servidor en producción. Una vez corregido el error la solución obvia es para el proceso servidor y volver a iniciarlo el servidor para que se cargue el nuevo código.

Erlang ofrece la posibilidad de cambiar el código que está ejecutando un proceso (por ejemplo un servidor) sin necesidad de detenerlo otros lenguajes o frameworks tienen otros mecanismo

El *runtime system* de Erlang guarda en cada momento dos versiones de cada módulo: la vieja y la actual cuando una nueva versión del módulo A se carga entonces esta se convierte en la actual, y la antigua versión actual pasa a ser la vieja

# Cambio de código en caliente

El mecanismo de ***software upgrade*** funciona de forma diferente para lllamadas intermódulo y para llamadas intramódulo

En Erlang hay dos maneras de llamar a una función

- llamada completamente cualificada `modulo:funcion(arg1, ..., argn)`
- llamada directa `funcion(arg1, ..., argn)`

  a funciones locales (definidas en el propio módulo) o importadas de otro módulo

Llamadas intermódulo: si una función f del módulo A llama a otra del módulo B y luego la definición de f en el módulo B se actualiza, las llamadas siguientes a f desde A corresponderán a la nueva versión de f

Demo `inter_upgrade.erl` y `other.erl`

Llamadas intramódulo: sólo las llamadas cualificadas se actualizarán

por tanto en los loops de servidores que se quieran actualizar se usan llamadas cualificadas
OjO para poder cualificar una llamada a f se debe exportar f

Demo `modtest2.erl`

Sólo existen las dos últimas versiones del código al introducir una nueva versión la versión vieja se ***purga***, y los cualquier proceso ejecutando esa versión más vieja se termina

Demo `modtest2.erl`

# Cambio de código en caliente

La directiva `-vsn/1` (por ejemplo `-vsn(1.0)`) permite asociar un número de versión al módulo (se asigna automáticamente en otro caso), útil para controlar versiones. Accesible usando `module_info/0` y `module_info/1`

```
> modtest2:module_info() .
> modtest2:module_info(attributes) .
```

# Cambio de código en caliente

¿Cuando se carga el código de un módulo? y por tanto se actualizan sus versiones actual y vieja?

- al llamar a una función de un módulo que aún no se ha usado el intérprete buscará el .beam del módulo correspondiente, empezando por la ruta '.', y si no está falla (no lo compila)
- compilando con el comando `c/1` de consola o con la función `compile:file/1` genera el .beam y lo carga
- cargando explícitamente el módulo llamando a la función `code:load_file/1`

Code Server es un proceso que es parte del kernel de Erlang y que se ocupa de gestionar la carga de código. En el módulo code encontramos funciones para manipularlo

- `code:get_path/0` devuelve una lista de rutas, llamada code path, que se recorrerán para buscar los .beam con los que cargar un módulo: se carga el primero que se encuentre desde el principio del code path nótese que "." es la primera ruta
- `code:add_patha(Ruta)`, `code:add_pathz(Ruta)` añade una ruta al principio o al final del code path por ejemplo se puede añadir una carpeta para parches al principio del code path

También se puede modificar el code path al abrir el intérprete mediante `erl -pa Path` o `erl -pz Path`

En la home del SO o en el Erlang root directory (`code:root_dir/0`) se puede crear un archivo .erlang con expresiones Erlang que se evalúan al iniciar la máquina virtual por ejemplo llamadas a `code:add_patha/1`

# Computación distribuida

Diseñar una aplicación como varios procesos que se ejecutarán en varias máquinas diferentes, posiblemente distribuidas geográficamente

## Ventajas

- El rendimiento  puede ser escalado bajo demanda: cuantas más maquinas añada más carga será capaz de soportar el sistema (pej un servidor)
- Tolerancia a fallos por medio de replicación: si una de las máquinas cae el resto todavía podrán responder a las peticiones
- Acceso transparente a recursos remotos: necesario en organizaciones internacionalizadas con sus recursos dispersos geográficamente (pej el estado de los almacenes en una empresa de comercio global)
- Permite compartir recursos de cómputo entre organizaciones

## Desventajas

- Complejidad del diseño
- Es más difícil garantizar la seguridad porque los datos que maneja la aplicación están moviéndose a través de alguna red

Erlang ofrece soporte nativo para la computación distribuida

# Computación distribuida

**Nodo Erlang** un runtime system de Erlang al que se le ha dado un nombre

**Host** una máquina que forma parte del sistema distribuido

Varios nodos pueden ejecutarse en el mismo host, pero también pueden ejecutarse en varios

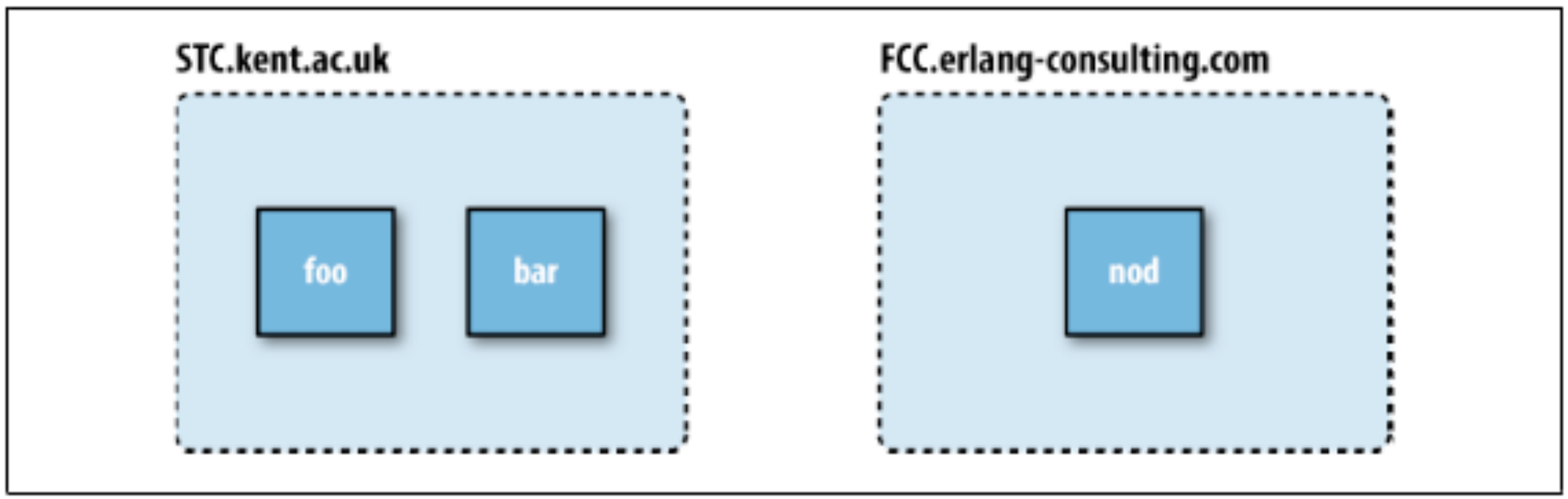



*Figure 11-1. Three nodes running on two hosts*

**Demo** `dist.erl` dos nodos en el mismo host

# Computación distribuida

`spawn(Node, Module, Function, Args)` inicia un proceso en otro nodo

`node/0` devuelve el nombre del nodo actual

Transparencia (casi) en el ejemplo mandamos mensajes usando el Pid sin importarnos que el proceso destino esté o no en el nodo actual.
Sin embargo para procesos con nombre (`register/2`) se usa `{RegName, Node} ! Msg`

```
-module(dist) .
-export([t/1]) .

t(From) -> From ! node() .
```

Sólo los nodos vivos (alive) pueden comunicarse con otros, los nodos vivos son los que tienen nombre por:
- haber iniciado la máquina con `erl -sname <nombre>` para redes locales
- haber iniciado la máquina con `erl -name <nombre>` para redes globales
- `net_kernel:start([nombre])` convierte el proceso actual en un nodo vivo con ese nombre
- `net_kernel:stop/0` hace que el nodo actual deje de estar vivo

`is_alive/0` devuelve `true` o `false` según el nodo actual esté vivo o no

# Computación distribuida

Seguridad sólo en redes seguras y entre nodos fiables, se basa en un sistema de magic cookies

- se genera una aleatoria en `~/.erlang.cookie`
- se puede asignar con `erl -sname foo -setcookie blah`
- también mediante `erlang:set_cookie(node(), blah)`
- `erlang:get_cookie/0` devuelve la actual

Sólo nodos con la misma magic cookie se pueden comunicar si no especificamos una cookie todos los nodos en la misma máquina podrán comunicarse porque compartirán `~/.erlang.cookie`

En cuanto dos nodos se comunican entonces pueden ejecutar cualquier instrucción en el otro nodo: en particular `spawn(YourNode, os, cmd, ["rm -rf *"])`

Además cookies se lanzan sin encriptar: dependemos de la seguridad de la red

Código ni el fuente `.erl` ni los `.beam` se transfieren automáticamente entre los nodos, por lo que al hacer `spawn/4` en un nodo remoto debemos estar seguros que de tiene acceso al `.beam` del código a ejecutar

# Computación distribuida

Conectando nodos una vez los nodos se han conectado, por tener la misma cookie, siguen conectados aunque alguno de ellos cambie su cookie

```
$ erl -sname foo -setcookie fish
$ erl -sname bar -setcookie cake
(foo@STC)1> net_adm:ping('bar@STC').
(foo@STC)2> erlang:set_cookie(node(),cake).
(foo@STC)3> net_adm:ping('bar@STC').
(foo@STC)4> erlang:set_cookie(node(),fish).
(foo@STC)5> net_adm:ping('bar@STC').
```

**`net_adm:ping(Node)`** intenta conectar con otro nodo, devuelve **`pang`** en caso de fallo y **`pong`** en caso de éxito. Útil para probar la conexión entre nodos.

**`net_kernel:connect_node(Node)`** intenta conectar con otro nodo, devuelve true o false indicando el éxito

La conexión entre nodos es transitiva N nodos implica N*(N-1)/2 conexiones

- Para evitar la proliferación de conexiones entre nodos que no trabajan juntos se usan los nodos ocultos (hidden) **`erl -sname foo -hidden`**

ejemplo: gamma hidden, alpha y beta normales

```
(alpha@STC)1> net_kernel:connect('beta@STC').
(alpha@STC)2> net_kernel:connect('gamma@STC'). % beta no se conecta con gamma
(alpha@STC)3> nodes().                         % nodos no ocultos conectados
(alpha@STC)4> nodes(hidden).                   % nodos ocultos conectados
(alpha@STC)5> nodes(connected).                % todos los nodos conectados
(beta@STC)3> nodes(connected).
(gamma@STC)3> nodes().
```

# Computación distribuida

## Enlace y monitorización distribuida

`spawn_link(Node, Module, Function, Args)` inicia un proceso en otro nodo y enlace el proceso creado con el proceso invocador si se pierde la conexión entre los nodos se mandará una exit signal con `noconnection` como razón

`monitor_node(Node, true)` inicia la monitorización de `Node`, de forma que un mensaje `{nodedown, Node}` se enviará al proceso invocador cuando `Node` termine su ejecución (un mensaje por cada llamada a `monitor_node(Node, true)`)

`monitor_node(Node, false)` termina la monitorización de `Node`