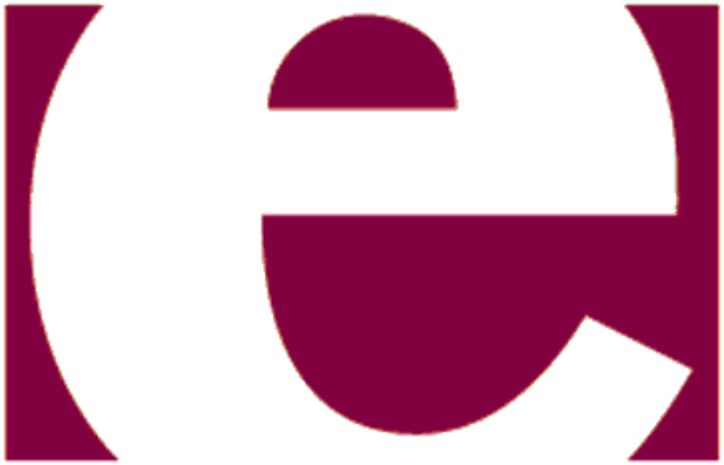
# Instituto Superior de Engenharia do Porto

## Licenciatura em Engenharia Informática

### Projecto de Licenciatura



# A Plataforma Erlang/OTP


**André Ferreira**
**i990280**

**Orientador: Eng.º Paulo Ferreira**



**versão 1.0, Junho de 2004**

# Agradecimentos e Motivações

Com o chegar deste último semestre dum longo percurso de cinco anos, chega-se também perto do fim de um ciclo, com muitos momentos bons, e outros não tão bons mas que foram ultrapassados. Está-se prestes a deixar a vida académica e a ingressar no mundo do trabalho, o começo de uma nova etapa na vida de cada um de nós.

Escolhi este projecto pela minha atracção pelo mundo das telecomunicações, uma área na qual gostava de trabalhar findo o curso. Achei que podia aprender mais alguma coisa, sobre uma plataforma desenhada em exclusivo para esse propósito, e que tal me podia ser útil no futuro.

Gostaria de agradecer ao meu orientador, Eng.º Paulo Ferreira pela prontidão e disponibilidade, empenho, sugestões dadas e interesse demonstrado durante a realização deste relatório.

Gostaria ainda de agradecer o apoio e amizade de colegas e amigos, e em particular da minha família.

# 1. Introdução

A procura de aplicações de tempo real de alto desempenho e que atendam ao mercado da indústria de telecomunicações deu início à pesquisa e desenvolvimento de uma arquitectura promissora na área dos sistemas distribuídos. O Erlang/OTP é um ambiente de desenvolvimento para a construção de sistemas distribuídos de tempo real de alto desempenho. O sistema é baseado na linguagem de programação Erlang e consiste num sistema *run-time* Erlang, uma biblioteca de componentes reutilizáveis, e um conjunto de princípios de desenvolvimento para programas Erlang. A arquitectura Erlang/OTP já é utilizado em alguns produtos tais como o *AXD301, DWOS, A910* e o Anx da empresa Ericsson.

# 2. A linguagem de programação Erlang

Ao contrário da maioria das linguagens de programação actuais, a linguagem Erlang é uma linguagem funcional, e as suas primeiras versões foram escritas em *Prolog*. Os avanços e estudos dos paradigmas da programação funcional mostraram que essa forma de programação podia ser usado para resolver vários problemas do mundo das telecomunicações. Várias primitivas da linguagem Erlang prevêem facilidades para a construção de programas de tempo real. O mecanismo de detecção de erros permite construir software com tolerância a falhas. Primitivas de execução de código permitem que o código dos processos seja actualizado sem que o sistema tenha que ser interrompido (*non stop systems*).

O processo de actualização das variáveis no Erlang é feito apenas através de trocas de mensagens, isso permite que uma aplicação que esteja a ser executada num computador possa ser facilmente portada para um ambiente distribuído. Os programas são escritos usando funções, o que os torna mais

sucintos e fiáveis. Algumas das principais características da linguagem Erlang são:

- **Sintaxe declarativa** – os programas Erlang são escritos utilizando a sintaxe das linguagens declarativas;

- **Concorrente** – a linguagem baseia-se num modelo assíncrono de troca de mensagens;

- **Tempo real** – a linguagem Erlang foi construída para construção de sistemas onde o tempo de execução é um factor crítico. As primitivas da linguagem facilitam esse tipo de programação.

- **Operação contínua** – os processos podem ser actualizados sem que todo o sistema tenha que ser interrompido.

- **Robustez** – os mecanismos de detecção e recuperação de erros possibilitam a criação de programas robustos e de alto desempenho.

- **Gestão de memória** – a linguagem Erlang tem um processo que se encarrega de fazer a gestão de memória, o *garbage colector*. O processo é o responsável pela alocação e desalocação de memória, de acordo com a execução dos processos.

- **Distribuição** – a linguagem Erlang não usa memória partilhada, toda a troca de informações entre os processos é feita através de trocas de mensagens. Isso permite que os programas sejam facilmente colocados num sistema distribuído.

- **Integração** – a linguagem Erlang permite que sejam executadas funções que foram escritas em outras linguagens. Todo o processo é feito de forma transparente pelo programador.

# 3. Arquitectura Erlang/OTP

O Erlang foi criado no Laboratório de Ciências da Computação de Ellemtel, do grupo Ericsson. O objectivo inicial da pesquisa era criar uma linguagem de programação mais flexível que estivesse de acordo com as necessidades da indústria de telecomunicações. Algumas características da arquitectura Erlang em termos de aplicação são:

- **Concorrência**: prevê a execução simultânea de vários processos;
- **Robustez**: os erros podem ser capturados e tratados de forma que a aplicação nunca seja interrompida devido a eles.
- **Distribuídos**: os sistemas feitos em Erlang podem e devem ser distribuídos por vários computadores, tanto por questões de distribuição de aplicações como para tornar as aplicações mais robustas e eficientes.

A *Open Telecom Platform* (OTP) é um sistema de desenvolvimento desenhado para construir e executar sistemas de telecomunicações. Como

pode ser visto pela figura 1, o sistema OTP é aquilo a que se chama uma "plataforma *Middleware*", desenhada para ser executada por cima de um sistema operativo convencional.

O OTP é portável entre diferentes tipos de sistemas operativos, tais como Linux, FreeBSD, Solaris, OS-X, Windows, VxWorks, etc. Isto é possível porque a plataforma OTP é executada através do sistema *run-time* do Erlang, que é uma máquina virtual desenhada para executar o código gerado pelo compilador do Erlang, que compila código Erlang em sequências de instruções para interpretadores "*word threaded*" de 32 bits. Este compilador tem o nome de BEAM. Para uma maior eficiência, os programas escritos em Erlang podem ainda ser compilados para código nativo usando o compilador HIPE.

O sistema *run-time* do Erlang disponibiliza muitos dos serviços que são normalmente oferecidos pelos sistemas operativos, o que o torna bastante complexo. Todos os processos são geridos por este sistema *run-time*. Isto significa que mesmo quando há várias dezenas de milhar de processos a serem executados pelo sistema *run-time* do Erlang, o sistema operativo no qual o sistema está a ser executado só "vê" um processo: o sistema *run-time* do Erlang.

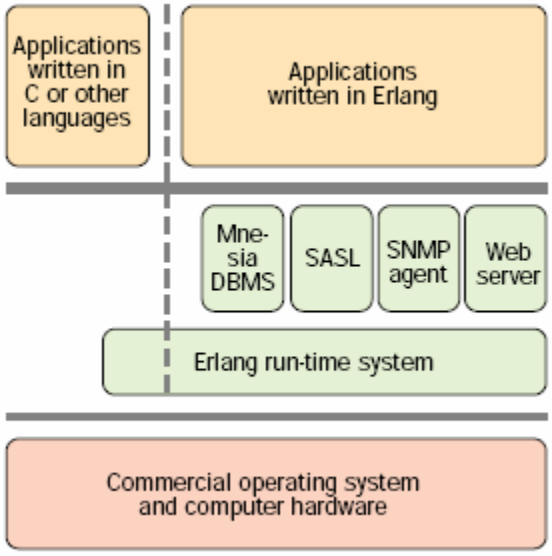
Por outro lado, o compilador Erlang é muito simples, comparado com compiladores de outras linguagens. O acto de compilação é uma simples tradução de código Erlang em primitivas apropriadas da máquina virtual.

No Erlang cada processo possui a sua própria área de memória. A comunicação entre os processos é feita através de troca de mensagens. Um mecanismo nativo de tratamento de erros identifica e recupera os processos que podem ter dado erro. É possível actualizar o código de um processo mesmo que este esteja em execução. Os programas Erlang podem ser executados numa estação de trabalho simples ou podem ser distribuídos por vários nós. A comunicação entre os processos e a detecção de erros têm as mesmas propriedades ao serem executados num nó simples ou num sistema distribuído.

## 3.1. Visão Geral

Camada de baixo

É constituída pelos computadores, ou seja o Hardware. Pode englobar soluções comercias ou soluções construídas em casa, dependendo dos custos de produção. Esta camada normalmente é constituída por muitos computadores de diversos tipos.




• Figura 1 – Visão da Arquitectura Erlang/OTP

Camada intermédia

É a camada que suporta os requisitos de comunicação, através de um DBMS (*Data Base Management System*) robusto, de tempo real e distribuído. Fornece suporte para manusear software e reportar eventos. Está dotado de um agente SNMP, um servidor web, uma biblioteca de funções para inter operacionalidade entre aplicações escritas em C e Erlang.

Camada de Topo

Todas as aplicações têm acesso ao DBSM e ao SASL. O agente SNMP e o servidor web também podem invocar funções que forem disponibilizadas pelas aplicações desta camada.

## 3.2. Target Systems

Uma aplicação OTP que consiste num sistema de controlo com um, ou mais processadores, pode ser configurado como sendo um target system. Se o sistema compreender mais que um processador, então esses processadores têm de ser capazes de comunicar entre si. Alguns processadores supervisionam e controlam o sistema através de SASL (*System's Architecture Support Libraries*), que é o componente central do OTP.

Um mecanismo de comunicação junta todos os processadores ao sistema distribuído. Se for necessário, os processadores e o mecanismo de comunicação podem ser duplicados.

O OTP trata de todos os requerimentos de telecomunicações para o sistema de controlo (tempo-real, tolerância a falhas, *upgrades* de software em tempo real, distribuição, etc.), o que permite que os programadores se concentrem nos aspectos mais específicos do projecto em que estão a trabalhar. Uma grande vantagem do OTP é que é independente do sistema operativo.

## 3.3. Máquina Virtual

Actualmente, várias linguagens de programação existentes têm um objectivo principal que é o de gerar código que seja independente da plataforma. Isso é possível porque nessas linguagens os programas são compilados para uma estrutura de dados intermediária, chamada *byte code*, e posteriormente são executados/interpretados por uma estrutura de software que é conhecida como máquina virtual, e não pelo hardware em particular. A máquina virtual é responsável pela execução dos programas e por torná-los independente da plataforma. O Erlang possui um funcionamento semelhante. Existem e estão a ser criadas máquinas virtuais para várias plataformas.

A máquina virtual de Erlang funciona por cima do sistema operativo. O sistema *run-time* do Erlang funciona frequentemente como um único processo no sistema operativo. A máquina virtual de Erlang fornece o seguinte suporte para programas em Erlang:

- Uma interface consistente com o sistema operativo em qualquer plataforma.
- Alocação de memória e "*garbage collection*" em tempo real, que elimina de forma eficaz fugas de memória.
- Suporte a milhares de tarefas em simultâneo.
- Cooperação transparente entre todos os computadores do sistema.
- Posição e encapsulamento de erros em tempo de execução.
- A supervisão do código em tempo de execução à medida que vai sendo carregado ou substituído, e quando estiver a ser ligado.

A estrutura da máquina virtual do Erlang permite que seja incorporada facilmente em novos sistemas operativos. A máquina virtual do Erlang é o único *building block* que não foi escrito em Erlang

### 3.4. Plataforma Aberta

Um dos objectivos do Erlang/OTP era criar uma plataforma aberta tanto a nível de hardware como de software, ponto crítico no desenvolvimento de sistemas de telecomunicações. O OTP (*Open Telecom Plataform* - Plataforma de Telecomunicações Aberta), consiste num sistema *run-time* Erlang, vários componentes prontos a serem usados, e um conjunto de padrões de desenvolvimento para programas em Erlang. Como o Erlang é a base do OTP, o termo Erlang/OTP é usado em vez de OTP.

### 3.5. Alta Ajuda no Desenvolvimento

A ideia embutida no ciclo de desenvolvimento utilizando o Erlang é que os sistemas sejam vistos como um conjunto de serviços. A ideia é que a aplicação seja construída usando os componentes e bibliotecas existentes, aumentando a reutilização e produtividade no ciclo de desenvolvimento. Os processos são implementados usando regras, que são formalizações de padrões de desenvolvimento com suporte nativo para detecção e manipulação de erros, *trace* e políticas de actualização de código.

# 4. Princípios da Programação em OTP

Este conjunto de princípios instrui os programadores a estruturar o código Erlang em termos de processos, módulos e directórios, de forma a que esse mesmo código se torne mais fácil de entender.

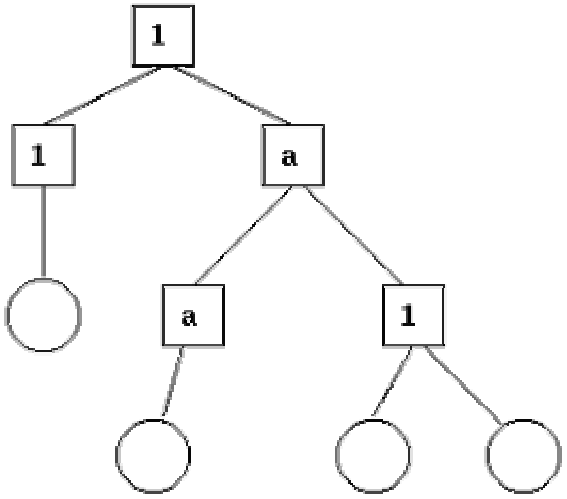
## 4.1. Arvores de Supervisão

Trata-se de um conceito básico na plataforma Erlang/OTP. É um modelo de estruturação de processos baseado na ideia de *workers* (trabalhadores) e *supervisors* (supervisores).

Os *workers* são processos que executam operações de computação, ou seja, executam o trabalho que se pretende fazer.

Os *supervisors* são processos que monitorizam o trabalho dos *workers*. Um supervisor pode reiniciar um *worker* caso se justifique.

A árvore de supervisão é um arranjo hierárquico de código em *workers* e *supervisors* que torna possível desenhar e programar sistemas tolerantes a falhas.




• Figura 2 - Arvore de Supervisão

Na figura anterior os quadrados representam supervisores e os círculos representam os *workers*.

## 4.2. Comportamentos

Numa árvore de supervisão, a maioria dos processos tem uma estrutura semelhante, e seguem padrões semelhantes. A única diferença entre os supervisores é quais os processos filhos que monitorizam. E grande parte dos *workers* são servidores numa relação cliente – servidor, máquinas de estados finitos ou *handlers* de eventos, como *logger* de erros.

Comportamentos (Behaviours) são os nomes dados às formalizações de padrões comuns, como os referidos acima. Pretende-se dividir o código de um processo em duas partes: uma genérica que vai ser o módulo de comportamento, e uma parte específica que vai ser um módulo de *callback*.

O módulo de comportamentos faz parte do Erlang/OTP. Para implementar um processo como supervisor, um programador tem apenas de implementar o módulo de *callback* que deve exportar um conjunto predefinido de funções, chamadas **funções de *callback***.

Em seguida temos um exemplo para ilustrar como o código pode ser dividido em duas parte de acordo com o que foi referido acima. Neste programa escrito em Erlang, um servidor mantêm um conjunto de canais. Outros processos podem alocar ou libertar os canais através da invocação das funções `alloc/0` e `free/0`.

```
-module(ch1).
-export([start/0]).
-export([alloc/0, free/1]).
-export([init/0]).

start() ->
   spawn(ch1, init, []).

alloc() ->
   ch1 ! {self(), alloc},
   receive
      {ch1, Res} ->
         Res
   end.

free(Ch) ->
   ch1 ! {free, Ch},
   ok.

init() ->
   register(ch1, self()),
   Chs = channels(),
   loop(Chs).

loop(Chs) ->
   receive
      {From, alloc} ->
         {Ch, Chs2} = alloc(Chs),
         From ! {ch1, Ch},
         loop(Chs2);
      {free, Ch} ->
         Chs2 = free(Ch, Chs),
         loop(Chs2)
   end.
```

O código do servidor pode ser reescrito numa parte genérica:

```
-module(server).
-export([start/1]).
-export([call/2, cast/2]).
-export([init/1]).
```

```
start(Mod) ->
   spawn(server, init, [Mod]).

call(Name, Req) ->
   Name ! {call, self(), Req},
   receive
      {Name, Res} ->
         Res
   end.

cast(Name, Req) ->
   Name ! {cast, Req},
   ok.

init(Mod) ->
   register(Mod, self()),
   State = Mod:init(),
   loop(Mod, State).

loop(Mod, State) ->
   receive
      {call, From, Req} ->
         {Res, State2} = Mod:handle_call(Req, State),
         From ! {Mod, Res},
         loop(Mod, State2);
      {cast, Req} ->
         State2 = Mod:handle_cast(Req, State),
         loop(Mod, State2)
   end.
```

E numa parte específica, a de *callback*:

```
-module(ch2).
-export([start/0]).
-export([alloc/0, free/1]).
-export([init/0, handle_call/2, handle_cast/2]).

start() ->
   server:start(ch2).

alloc() ->
   server:call(ch2, alloc).

free(Ch) ->
   server:cast(ch2, {free, Ch}).

init() ->
   channels().

handle_call(alloc, Chs) ->
```

```
    alloc(Chs). % => {Ch,Chs2}

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

É de salientar o seguinte:

- O código da parte genérica pode ser reutilizado para construir muitos servidores diferentes.
- O nome do servidor, neste exemplo é o átomo `ch2`, é escondido dos utilizadores das funções de cliente, ou seja, pode ser alterado sem as afectar.
- O protocolo também está escondido, o que é uma boa estratégia de programação, pois permite alterar o protocolo sem ter de alterar o código das funções de interface.
- Podemos expandir as funcionalidades do servidor sem ter de alterar o seu nome nem o módulo de *callback*.

De uma maneira muito simplificada, o módulo `server` corresponde ao comportamento OTP denominado **gen_server**. Os comportamentos OTP standards são:

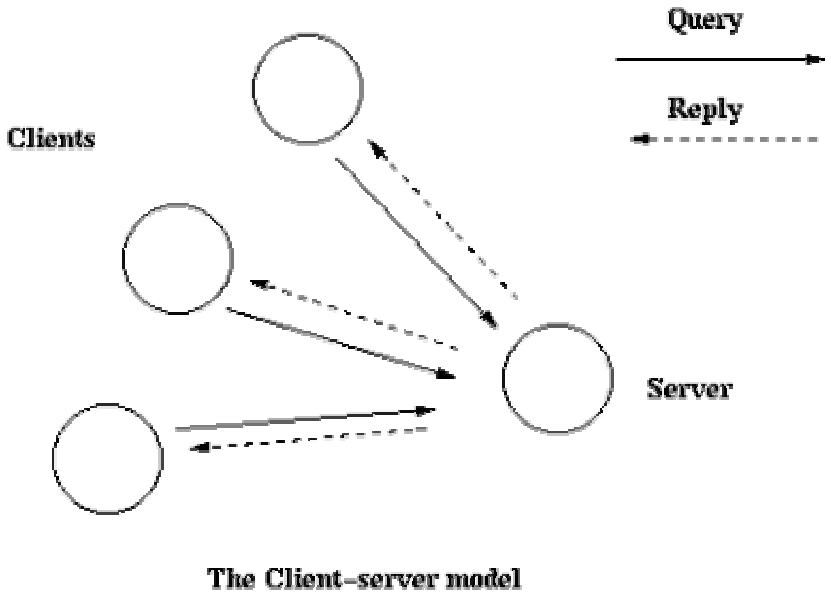
- gen_server – para implementar o servidor de uma relação cliente – servidor.
- gen_fsm – para implementar uma máquina de estados finitos.
- gen_event – para implementar funcionalidade de manuseamento de eventos.
- Supervisor – para implementar um supervisor numa árvore de supervisão.

Apesar do código escrito sem utilização de comportamentos poder ser mais eficiente, a sua utilização torna mais fácil a sua leitura e compreensão por outros programadores. Neste campo é muito importante a capacidade de se poder gerir todas as aplicações de um sistema de uma maneira mais simples. As estruturas de programação **ad hoc** podem ser mais eficientes mas são sempre mais difíceis de compreender.

## 4.3. O Comportamento gen_server

### 4.3.1. Princípios Cliente – Servidor

O modelo cliente – servidor é caracterizado por um servidor central e um vasto conjunto de clientes. Este modelo é normalmente usado para operações de manutenção de recursos, em que vários clientes diferentes pretendem partilhar um recurso comum. O servidor torna-se responsável por gerir esse recurso.




• Figura 3 - Modelo Cliente - Servidor

### 4.3.2. Exemplo

O exemplo do servidor dado anteriormente pode ser reescrito usando o comportamento gen_server, resultando um novo módulo de *callback*:

```
-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}.
```

### 4.3.3. Iniciar um gen_server

No exemplo anterior, o gen_server é iniciado chamando a função `ch3:start_link()`.

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

Esta função chama a função `gen_server:start_link/4` que faz um `spawn` e liga-se ao novo processo, um gen_server.

O primeiro argumento, `{local, ch3}`, especifica o nome, que neste caso é `Ch3`. Se o nome for omitido, o gen_server não é registado, pelo que se terá de usar o seu pid.

O segundo argumento, `Ch3`, é o nome do módulo de *callback*. Neste caso, as funções de interface `(start_link, alloc e free)` estão localizadas no mesmo módulo que as funções de *callback* `(init, handle_call e handle_cast)`. Isto é normalmente uma boa prática de programação.

O terceiro argumento, `[]`, é um termo que é passado para a função `init`. Neste caso não é necessário.

O quarto argumento, `[]`, é uma lista de opções.

Se o registo do nome for bem sucedido, o processo gen_server chama a função de *callback* `ch3:init([])` que é suposto retornar um tuplo `{ok,`

`State}`, onde `State` é o estado interno do gen_server. Neste caso é o estado dos canais disponíveis. É de fazer notar que a função `gen_server:start_link` é síncrona, ou seja, não retorna nada até que o gen_server seja iniciado e esteja pronto para receber pedidos.

### 4.3.4. Pedidos – Chamadas Síncronos

O pedido síncrono `alloc()` é implementado usando a função `gen_server:call/2`.

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

O pedido é feito através de uma mensagem que é enviada ao gen_server. Quando esse pedido é recebido, o gen_server chama a função `handle_call(Request, From, State)` que deve retornar um tuplo `{reply, Reply, State1}`. `Reply` é a resposta que deve ser enviada ao cliente. `State1` é o novo estado do gen_server.

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

Neste caso a resposta é o canal alocado `Ch` e o novo estado são os canais restantes, `Chs2`. A chamada de `ch3:alloc()`, além de retornar os valores referidos, irá ficar à espera de novos pedidos, com a lista de canais disponíveis actualizada.

### 4.3.5. Pedidos – Chamadas Assíncronas

O pedido assíncrono `free(Ch)` é implementado usando a função `gen_server:cast/2`.

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

`Ch3` é o nome do gen_server. `{free, Ch}` é o pedido actual. O pedido é efectuado através de uma mensagem que é enviada ao gen_server. Quando o pedido é recebido o gen_server chama a função `handle_cast(Request, State)` que retorna um tuplo `{noreply, State1}`. `State1` é o novo valor para o estado do servidor.

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

Neste exemplo, o novo estado é a lista de canais actualizada, `Chs2`. O gen_server fica pronto para receber novos pedidos.

#### 4.3.6. Terminar

Se o gen_server for parte de uma arvore de supervisão não é preciso nenhuma função de stop, pois irá ser terminado automaticamente pelo seu supervisor através da chamada de `exit(Pid, shutdown)`.

O processo gen_server não apanha os sinais de exit. Se for necessário fazer alguma coisa antes de terminar, este processo tem de ser instruído a apanhar esses sinais na função `init` e deve ser implementada uma nova função `terminate(Reason, State)` para fazer o que for preciso.

Se o gen_server não fizer parte de uma árvore de supervisão, uma função de stop pode ser útil. O gen_server irá chamar automaticamente a função `terminate(Reason, State)` se alguma das funções de *callback* retornar `{stop, Reason, State2}` em vez de `{reply,...}` ou `{noreply,...}`.

#### 4.3.7. Tratamento de outras mensagens

Se o gen_server for capaz de receber outras mensagens que não sejam pedidos, a função de *callback* `handle_info(Info, State)` tem de ser implementada de forma a poder tratar essas mensagens.

## 4.4. O Comportamento gen_fsm

### 4.4.1. Máquinas de Estado Finitas

Uma máquina de estados finitos, FSM (*Finite State Machine*), pode ser descrita como um conjunto de relações do tipo:

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

Uma relação deste tipo pode ser interpretada como: *Se estamos no estado S e ocorre o evento E, então devemos executar a acção A e transitar para o estado S'*.

Numa FSM, implementada usando o comportamento gen_fsm, as regras de transição de estados são escritas através de funções Erlang, de acordo com a convenção seguinte:

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

### 4.4.2. Exemplo

Uma porta com uma fechadura electrónica pode ser vista com uma máquina de estados finitos. No estado inicial a porta está fechada. Quando alguém pressiona um botão ocorre um evento. Se a sequencia de botões premidos corresponder ao código para abrir a porta, esta abre-se e mantêm-se aberta por um período tempo (a considera, 30 segundos). Se a sequência de botões premidos ainda estiver incompleta a porta mantém-se fechada e esperamos que outro botão seja premido. Se a sequência de botões premidos estiver completa e estiver errada, começamos de novos e esperamos que um novo botão seja premido. Implementando a FSM usando o comportamento gen_fsm resulta no seguinte módulo de *callback*:

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

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

```
start_link(Code) ->
  gen_fsm:start_link({local, code_lock}, code_lock, Code, []).

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

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

closed({button, Digit}, {SoFar, Code}) ->
  case [Digit|SoFar] of
    Code ->
      do_open(),
      {next_state, open, {[], Code}, 3000};
    Incomplete when length(Incomplete)<length(Code) ->
      {next_state, closed, {Incomplete, Code}};
    _Wrong ->
      {next_state, closed, {[], Code}};
  end.

open(timeout, State) ->
  do_close(),
  {next_state, closed, State}.
```

Iniciar um gen_fsm

No exemplo anterior, o gen_fsm é iniciado quando se chama a função code_lock:start_link(Code):

```
start_link(Code) ->
  gen_fsm:start_link({local, code_lock}, code_lock, Code, []).
```

Start_link chama a função `gen_fsm:start_link/4`. Esta função faz `spawn` e liga-se ao processo criado, um gen_fsm.

- O primeiro argumento, `{local, code_lock}`, especifica o nome, que neste caso é registado localmente como `code_lock`.
- O segundo argumento, `code_lock`, é o nome do módulo de *callback*, ou seja, o módulo onde se encontram as funções de *callback*.
- O terceiro argumento, `Code`, é um termo que é passado à função de *callback* `init`. Aqui, `init`, recebe o código correcto para abrir a porta como dados de entrada.
- O quarto argumento, `[]`, é uma lista de opções.

Se o registo do nome tiver sucesso, o processo gen_fsm criado irá chamar a função de *callback* `code_lock:init(Code)`. É esperado que essa função retorne um tuplo `{ok, StateName, StateData}`, onde `StateName` é o nome

do estado inicial do gen_fsm. Neste caso é fechada, se assumirmos que a porta se encontra fechada no início. `StateData` é o estado interno do gen_fsm, que neste caso é a sequência de botões premidos até ao momento, e o código correcto para abrir a porta.

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

É de salientar que a função `gen_fsm:star_link` é síncrona, não retorna nada até que o processo gen_fsm tenha sido inicializado e esteja pronto a receber pedidos.

### 4.4.3. Notificações de Eventos

As funções que notificam o sistema (nesta caso, a fechadura) que ocorreram eventos, são implementadas recorrendo à função `gen_fsm:send_event/2`.

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

O parâmetro `code_lock` é o nome do processo gen_fsm e tem de estar de acordo com o nome usado para o iniciar. `{button, Digit}` é o evento actual. O evento é enviado através de um mensagem para o processo gen_fsm. Quando é recebido, o gen_fsm chama a função `StateName(Event, StateData)` que retorna um tuplo `{next_state, StateName1, StateData1}`. `StateName` é o nome do estado corrente e `StateName1` é o nome do próximo estado. `StateData1` é o novo valor para os dados de estado do gen_fsm.

```
closed({button, Digit}, {SoFar, Code}) ->
  case [Digit|SoFar] of
    Code ->
      do_open(),
      {next_state, open, {[], Code}, 30000};
    Incomplete when length(Incomplete)<length(Code) ->
      {next_state, closed, {Incomplete, Code}};
```

```
        _Wrong ->
            {next_state, closed, {[], Code}};
    end.

open(timeout, State) ->
    do_close(),
    {next_state, closed, State}.
```

Se a porta está fechada e um botão foi premido, a sequência de botões premidos até ao momento é comparada com o código correcto da fechadura, e, dependendo do resultado, a porta é aberta e o processo gen_fsm passa ao estado aberta, ou a porta mantém-se fechada.

### 4.4.4. Timeouts

Quando é dado um código correcto, a porta abre-se e é retornado um tuplo da função `closed/2`: `{next_state, open, {[], Code}, 30000}`.

O valor 30000 é o valor de timeout em milissegundos. Depois desse tempo se esgotar, ocorre um timeout e a função `StateName(timeout, StateData)` é invocada. Neste caso o timeout ocorre quando a porta se encontra aberta durante 30 segundos, e então fecha-se.

```
open(timeout, State) ->
    do_close(),
    {next_state, closed, State}.
```

### 4.4.5. Eventos de Todos Estados

Por vezes um evento pode ocorrer em qualquer estado do processo gen_fsm. Nesse caso, em vez de enviar uma mensagem através da função `gen_fsm:send_event/2` e se escrever uma clausula para tratar o evento em cada estado, envia-se a mensagem usando a função `gen_fsm:send_all_state_event/2` e é tratada com a função `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}.
```

### 4.4.6. Terminar

Se o processo gen_fsm fizer parte de uma árvore de supervisão não é necessário nenhuma função de stop, pois o processo irá ser terminado automaticamente pelo seu supervisor através da invocação da função `exit(Pid, shutdown)`.

O processo gen_fsm não trata sinais de `exit` por defeito, pelo que se for necessário executar algum código antes do processo terminar, deve-se indicar ao processo para apanhar esses sinais, na função `init`. Deve-se também criar outra função chamada `terminate(Reason, State)` para executar o código que se pretende antes do processo terminar.

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

...

terminate(shutdown, StateName, StateData) ->
   ..código a ser executado..
   ok.
```

Se o processo gen_fsm não faz parte de um árvore de supervisão, pode ser útil dispor de uma função de stop. O processo gen_fsm chama automaticamente a função de *callback* `terminate(Reason, StateName, StateData)` se alguma das outras funções de *callback* retornar um tuplo `{stop, Reason, State2}` em vez de `{next_state,...}`. Exemplo:

```
-module(code_lock).
...
-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.
```

#### 4.4.7. Tratar outras Mensagens

Se o processo gen_fsm for implementado para receber mais mensagens que não eventos, a função de `callback handle_info(Info, StateName, StateData)` tem de ser implementada de forma a poder tratar essas outras mensagens.

### 4.5. O Comportamento gen_event

#### 4.5.1. Princípios para tratar Eventos

No OTP, um gestor de eventos é um objecto conhecido para o qual os eventos podem ser enviados. Um evento pode ser um alarme, um erro ou alguma informação para ser adicionada aos *logs*.

Num gesto de eventos, EM (*Event Manager*) podem estar instalados zero, um ou vários *handlers* de eventos. Quando o EM é notificado sobre um evento, esse evento irá ser processado por todos os *handlers* instalados. Um EM é implementado sob a forma dum processo e cada *handler* de evento é implementado sob a forma de um módulo de *callback*.

### 4.5.2. Exemplo

O módulo de *callback* para um *handler* de eventos para escrever mensagens de erro na janela de terminal pode tomar o seguinte aspecto:

```
-module(terminal_logger).
-behaviour(gen_event).

-export([init/1, handle_event/2, terminate/2]).

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

handle_event(ErrorMsg, State) ->
   io:format("***Error*** ~p~n", [ErrorMsg]),
   {ok, State}.

terminate(_Args, _State) ->
   ok.
```

O módulo de *callback* para o *handler* de eventos para escrever mensagens de erros para ficheiros pode ser:

```
-module(file_logger).
-behaviour(gen_event).

-export([init/1, handle_event/2, terminate/2]).

init(File) ->
   {ok, Fd} = file:open(File, read),
   {ok, Fd}.

handle_event(ErrorMsg, Fd) ->
   io:format(Fd, "***Error*** ~p~n", [ErrorMsg]),
   {ok, Fd}.

terminate(_Args, Fd) ->
   file:close(Fd).
```

### 4.5.3. Iniciar um Gestor de Eventos

De acordo com o exemplo anterior, o gestor de eventos é iniciado da seguinte forma, a partir da *shell* do Erlang:

```
1> gen_event:start_link({local, error_man}).
{ok,<0.30.0>}
```

Esta função faz `spawn` e liga-se ao novo processo criado, um gestor de eventos. O argumento `{local, error_man}` especifica o nome, que neste caso está registado localmente como `error_man`. Se o nome for omitido, o gestor de eventos não será registado e terá que se usar o seu `pid` em seu lugar.

### 4.5.4. Adcionar um Handler de Evento

A partir da *shell* do Erlagn:

```
2> gen_event:add_handler(error_man, terminal_logger, []).
ok
```

Esta função envia uma mensagem ao gestor de eventos dizendo-lhe para adicionar um novo *handler* chamado `terminal_logger`. O gestor de eventos invoca a função de `callback terminal_logger:init([])`, onde o argumento [] é o terceiro argumento da função add_*handler*. Init retorna um tuplo `{ok, State}`, onde `State` é o estado interno do *handler*.

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

Neste exemplo, `init` não precisa de qualquer dado de entrada e como tal ignora esse argumento.

```
init(_Args) ->
   {ok, Fd} = file:open(File, read),
   {ok, Fd}.
```

### 4.5.5. Notificações sobre Eventos

Na *shell* do Erlang:

```
3> gen_event:notify(error_man, no_reply).
***Error*** no_reply
ok
```

O parâmetro `error_man` é o nome do gestor de eventos e `no_reply` é o próprio evento. O evento é enviado sob a forma de uma mensagem para o gestor de eventos. Quando esse evento é recebido, o gestor de eventos chama a função `handle_event(Event, State)` para cada um dos *handlers* instalados, pela ordem em que foram instalados. Esta função deve retornar um tuplo `{ok, State1}`, onde `State1` é o novo valor para o estado do *handler* de eventos.

Para o *handler* `terminal_logger`:

```
handle_event(ErrorMsg, State) ->
  io:format("***Error*** ~p~n", [ErrorMsg]),
  {ok, State}.
```

Para o *handler* `file_logger`:

```
handle_event(ErrorMsg, Fd) ->
  io:format(Fd, "***Error*** ~p~n", [ErrorMsg]),
  {ok, Fd}.
```

#### 4.5.6. Apagar um Handler de Eventos

```
4> gen_event:delete_handler(error_man, terminal_logger, []).
ok
```

Esta função envia uma mensagem ao gestor de eventos dizendo-lhe para apagar o *handler* `terminal_logger`. O gestor de eventos invoca então a função de *callback* `terminal_logger:terminate([], State)`, onde o argumento `[]` representa o terceiro argumento da função `delete_handler`. `Terminate` deve ser encarada como o oposto da função `init`.

Para o *handler* `terminal_logger`, não é necessário executar mais nenhum código:

```
terminate(_Args, _State) ->
  ok.
```

Para o *handler* `file_logger`, é necessário fechar o descriptor de ficheiros aberto na função `init`:

```
terminate(_Args, Fd) ->
  file:close(Fd).
```

#### 4.5.7. Terminar

Se o gestor de eventos fizer parte de um arvore de supervisão, não é necessário nenhuma função de stop, pois irá ser terminado automaticamente pelo seu supervisor através da função `exit(Pid, shutdown)`.

Um gestor de eventos também pode ser encerrado através da função, de acordo com o exemplo anterior:

```
> gen_event:stop(error_man).
ok
```

Quando um gestor de eventos é encerrado, é dada a possibilidade a cada *handler* instalado de executar algum código através da função `terminate/2`, da mesma forma como quando se apagava um *handler*.

### 4.6. Comportamento de Supervisor

#### 4.6.1. Princípios de Supervisão

Um supervisor é responsável por iniciar, terminar e monitorizar processos filhos. O objectivo principal é que o supervisor mantenha os processos filho sempre em funcionamento, iniciando um novo sempre necessário. A lista de processos filho a iniciar e monitorizar é dada por uma lista de especificações de processos filhos, que será abordada mais adiante.

### 4.6.2. Exemplo

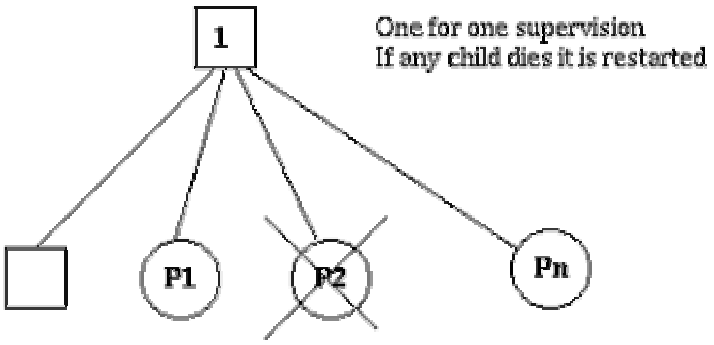
O módulo de *callback* para um supervisor iniciar o servidor apresentado no capítulo sobre o comportamento gen_server, pode-se parecer com o seguinte:

```
-module(ch_sup).
-behaviour(supervisor).

-export([start_link/0]).
-export([init/1]).

start_link() ->
   supervisor:start_link(ch_sup, []).

init(_Args) ->
   {ok, {{one_for_one, 1, 60},
        [{ch3, {ch3, start_link, []},
          permanent, brutal_kill, worker, [ch3]}]}}.
```

### 4.6.3. Estratégia de Reiniciação

#### 5.6.3.1 one_for_one

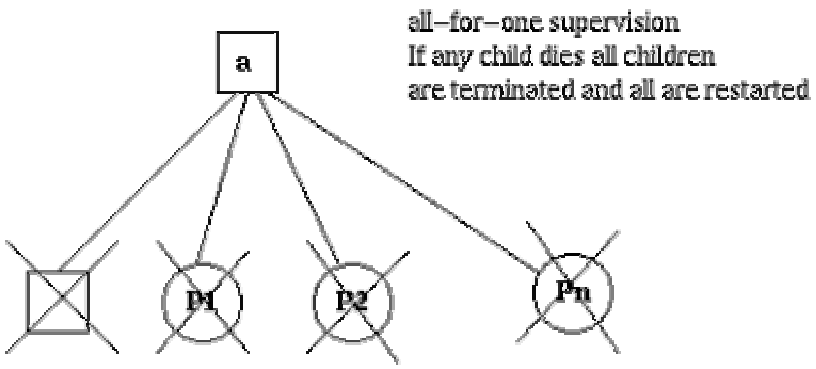
Se um processo filho terminar, apenas esse processo deve ser reiniciado.




• Figura 4 – Estratégia um por um

#### 5.6.3.2 one_for_all

Se um processo filho terminar, todos os outros processos filhos são terminados também, e então são todos reiniciados.




• Figura 5 – Estratégia um por todos

#### 5.6.3.3 rest_for_one

Se um processo filho terminar, o resto dos processos filhos, ou seja, todos os processos filhos criados depois do processo que terminou, são terminados também, e são reiniciados conjuntamente com o que tinha terminado.

### 4.6.4. Frequência Máxima de Reinícios

Os supervisores dispõem de um mecanismo interno para limitar o número de reinícios que podem ocorrer num dado intervalo de tempo. Esse limite é determinado de acordo com o valor dos parâmetros `MaxR` e `MaxT` retornados pela função de *callback* `init`:

```
init(...) ->
   {ok, {{RestartStrategy, MaxR, MaxT},
       [ChildSpec, ...]}}.
```

Se ocorrerem mais que `MaxR` reinícios durante o período de tempo `MaxT`, o supervisor termina todos os processos filho e em seguida termina ele próprio. A intenção do mecanismo de reinícios é prevenir situações em que um processo morre repetidamente pela mesma razão apenas com a intenção de reiniciado.

### 4.6.5. Especificação de Processos Filho

Em seguida é apresentada a definição de tipos para especificação de processos filhos:

```
{Id, StartFunc, Restart, Shutdown, Type, Modules}
  Id = term()
  StartFunc = {M, F, A}
    M = F = atom()
    A = [term()]
  Restart = permanent | transient | temporary
  Shutdown = brutal_kill | integer() &gt;=0 | infinity
  Type = worker | supervisor
  Modules = [Module] | dynamic
    Module = atom()
```

- `Id` é o nome usado pelo supervisor para identificar a especificação internamente.
- `StartFunc` define a função a ser chamada para iniciar o processo filho.
- `Restart` define quando é que um processo que terminou deve ser reiniciado:
  - Um processo filho permanente é sempre reiniciado;
  - Um processo filho temporário nunca é reiniciado;
  - Um processo filho *"transient"* é reiniciado apenas se terminar de forma anormal.
- `Shutdown` define a forma como um processo filho deve ser terminado:
  - `Brutal_kill` significa que o processo é terminado incondicionalmente usando a função `exit(Child, kill)`.
  - Um valor de timeout inteiro significa que o supervisor manda o processo filho terminar através da função `exit(Child, shutdown)` e depois espera por um sinal de exit. Se não o receber dentro de um período de tempo, o processo filho é terminado pela função `exit(Child, exit)`.
  - Se o processo filho é outro processo supervisor, deve ser colocado no estado `infinity` para dar tempo à sub árvore de processos supervisionados por si tempo para terminar.
- `Type` especifica se o processo filho é um *worker* ou um supervisor.

- `Modules` é uma lista com um elemento, `[Module]`, onde `Module` representa o nome do módulo de *callback*, se o processo filho for um supervisor, um gen_server ou um gen_fsm. Se for um gen_event, `Modules` deve ser `dynamic`.

A especificação de processos filhos para iniciar o servidor `ch3` deve ser algo como:

```
{ch3,
 {ch3, start_link, []},
 permanent, brutal_kill, worker, [ch3]}
```

A especificação de processos filhos para iniciar o gestor de eventos de exemplo usado no capítulo sobre o comportamento gen_event será algo como:

```
{error_man,
 {gen_event, start_link, [{local, error_man}]},
 permanent, 5000, worker, dynamic}
```

Iniciar um supervisor

```
{sup,
 {sup, start_link, []},
 transient, infinity, supervisor, [sup]}
```

Neste exemplo o supervisor é iniciado através da função `ch_sup:start_link()`:

```
start_link() ->
   supervisor:start_link(ch_sup, []).
```

A função `ch_sup:start_link` chama a função `supervisor:start_link/2`, que faz um `spawn` e liga-se ao novo processo, um supervisor. O primeiro argumento, `ch_sup`, é o nome do módulo de *callback*. O segundo argumento, `[]`, é um termo que é passado à função de *callback* `init`, que neste caso não é necessário.

Neste caso o supervisor não se encontra registado pelo que tem de se usar o seu pid. Pode ser especificado um nome chamando a função `supervisor:start_link({local, Name}, Module, Args)` ou `supervisor:start_link({global, Name}, Module, Args)`.

O novo processo supervisor chama então a função de *callback* `ch_sup:init([])` que é suposto retornar um tuplo `{ok, StartSpec}`:

```
init(_Args) ->
    {ok, {{one_for_one, 1, 60},
        [{ch3, {ch3, start_link, []},
          permanent, brutal_kill, worker, [ch3]}]}}.
```

O supervisor inicia então os seus processos filho de acordo com a especificação de processos filho.

#### 4.6.6. Adicionar um Processo Filho

Em complemento à árvore de supervisão estática, podemos adicionar dinamicamente processos filhos a um supervisor existente através da função `supervisor:start_child(Sup, ChildSpec)`, onde `Sup` é o pid ou nome do processo supervisor, e `ChildSpec` são as especificações do processo filho.

#### 4.6.7. Terminar um processo filho

Qualquer processo filho, seja estático ou dinâmico, pode ser terminado de acordo com as especificações de *shutdown*, através da função `supervisor:terminate_child(Sup, Id)`.

#### 4.6.8. Supervisores Um para Um Simples

Um supervisor com esta estratégia de reinícios é um supervisor **one_for_one** simplificado, onde todos os processos filho são instâncias do mesmo processo adicionadas de forma dinâmica.

```
-module(simple_sup).
-behaviour(supervisor).
```

```
-export([start_link/0]).
-export([init/1]).

start_link() ->
   supervisor:start_link(simple_sup, []).

init(_Args) ->
   {ok, {{simple_one_for_one, 0, 1},
        [{call, {call, start_link, []},
          temporary, brutal_kill, worker, [call]}]}}.
```

Quando o supervisor é iniciado, não vai iniciar nenhum processo filho. Em vez disso todos os processos filho são adicionados através do uso da função `supervisor:start_child(Sup, List)`

#### 4.6.9. Terminar

Uma vez que o supervisor faz parte de uma árvore de supervisão, irá ser automaticamente terminado pelo seu supervisor. Quando for "convidado" a terminar, vai primeiro terminar todos os seus processos filho pela ordem inversa de criação de acordo com lista de *shutdown* respectiva, e sem seguida termina ele próprio.

### 4.7. Os Módulos Sys e Proc_Lib

O módulo Sys contém funções para efectuar um *debug* simples de processos implementados usando comportamentos. Além dessas funções, inclui ainda algumas que funcionam em conjunto com funções do módulo Proc_Lib para implementar um tipo de processo especial, que obedece aos princípios da programação em OTP, mas sem recorrer ao uso de comportamentos.

#### 4.7.1. Debug Simples

O módulo Sys contém, como disse, funções para efectuar *debug*. Vamos recorrer ao exemplo da fechadura electrónica usada no capítulo sobre gen_fsm.

```
% erl
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> code_lock:start_link([1,2,3,4]).
{ok,<0.32.0>}
2> sys:statistics(code_lock, true).
ok
3> sys:trace(code_lock, true).
ok
4> code_lock:button(4).
*DBG* code_lock got event {button,4} in state closed
ok
*DBG* code_lock switched to state closed
5>  code_lock:button(3).
*DBG* code_lock got event {button,3} in state closed
ok
*DBG* code_lock switched to state closed
6> code_lock:button(2).
*DBG* code_lock got event {button,2} in state closed
ok
*DBG* code_lock switched to state closed
7> code_lock:button(1).
*DBG* code_lock got event {button,1} in state closed
ok
OPEN DOOR
*DBG* code_lock switched to state open
*DBG* code_lock got event timeout in state open
CLOSE DOOR
*DBG* code_lock switched to state closed
8> sys:statistics(code_lock, get).
{ok,[{start_time,{{2003,6,12},{14,11,40}}},
   {current_time,{{2003,6,12},{14,12,14}}},
   {reductions,333},
   {messages_in,5},
   {messages_out,0}]}
9> sys:statistics(code_lock, false).
ok
10> sys:trace(code_lock, false).
ok
11> sys:get_status(code_lock).
{status,<0.32.0>,
      {module,gen_fsm},
      [[{'$ancestors',[<0.30.0>]},
        {'$initial_call',{gen,init_it,
                          [gen_fsm,
                           <0.30.0>,
                           <0.30.0>,
                           {local,code_lock},
                           code_lock,
                           [1,2,3,4],
                           []]}}],
```

```
    running,
    <0.30.0>,
    [],
    [code_lock,closed,{[],[1,2,3,4]},code_lock,infinity]]}
```

### 4.7.2. O Processo Especial

Como referido no início deste capítulo, estas duas bibliotecas dispõem de funções que podemos usar para implementar um processo especial que obedece aos princípios da programação em OTP, sem recorrer ao uso de comportamentos. Esse processo deve:

- Ø Ser iniciado de maneira a fazer parte de uma árvore de supervisão.
- Ø Suportar as funcionalidades de *debug* da biblioteca Sys.
- Ø Tratar mensagens do sistema.

Mensagens do sistema são mensagens com significados especiais, usadas nas árvores de supervisão. Normalmente são pedidos de *trace*, pedidos para suspender ou resumir a execução de um processo. Os processos implementados recorrendo a comportamentos entendem estas mensagens automaticamente.

### 4.7.3. Exemplo

Este é o primeiro exemplo usado no capítulo sobre os princípios da programação em OTP, um servidor, desta feita implementado usando as bibliotecas sys e proc_lib, de forma a estar inserido numa árvore de supervisão.

```
-module(ch4).
-export([start_link/0]).
-export([alloc/0, free/1]).
-export([init/1]).
-export([system_continue/3, system_terminate/4,
        write_debug/3]).

start_link() ->
    proc_lib:start_link(ch4, init, [self()]).

alloc() ->
    ch4 ! {self(), alloc},
    receive
        {ch4, Res} ->
```

```
        Res
    end.

free(Ch) ->
    ch4 ! {free, Ch},
    ok.

init(Parent) ->
    register(ch4, self()),
    Chs = channels(),
    Deb = sys:debug_options([]),
    proc_lib:init_ack(Parent, {ok, self()}),
    loop(Chs, Parent, Deb).

loop(Chs, Parent, Deb) ->
    receive
        {From, alloc} ->
            Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                                    ch4, {in, alloc, From}),
            {Ch, Chs2} = alloc(Chs),
            From ! {ch4, Ch},
            Deb3 = sys:handle_debug(Deb2, {ch4, write_debug},
                                    ch4, {out, {ch4, Ch}, From}),
            loop(Chs2, Parent, Deb3);
        {free, Ch} ->
            Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                                    ch4, {in, {free, Ch}}),
            Chs2 = free(Ch, Chs),
            loop(Chs2, Parent, Deb2);

        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent,
                                  ch4, Deb, Chs)
    end.

system_continue(Parent, Deb, Chs) ->
    loop(Chs, Parent, Deb).

system_terminate(Reason, Parent, Deb, Chs) ->
    exit(Reason).

write_debug(Dev, Event, Name) ->
    io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

Exemplo de como as funções de *debug* da biblioteca sys podem ser usadas para ch4:

```
% erl
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> ch4:start_link().
```

```
{ok,<0.30.0>}
2> sys:statistics(ch4, true).
ok
3> sys:trace(ch4, true).
ok
4> ch4:alloc().
ch4 event = {in,alloc,<0.25.0>}
ch4 event = {out,{ch4,ch1},<0.25.0>}
ch1
5> ch4:free(ch1).
ch4 event = {in,{free,ch1}}
ok
6> sys:statistics(ch4, get).
{ok,[{start_time,{{2003,6,13},{9,47,5}}},
    {current_time,{{2003,6,13},{9,47,56}}},
    {reductions,109},
    {messages_in,2},
    {messages_out,1}]}
7> sys:statistics(ch4, false).
ok
8> sys:trace(ch4, false).
ok
9> sys:get_status(ch4).
{status,<0.30.0>,
       {module,ch4},
       [[{'$ancestors',[<0.25.0>]},{'$initial_call',{ch4,init,[<0.25.0>]}}],
        running,
        <0.25.0>,
        [],
        [ch1,ch2,ch3]]}
```

### 4.7.4. Iniciar o Processo

Para iniciar o processo usa-se uma função do módulo proc_lib. Há várias possíveis, chamadas `spawn_link`, com 3 ou 4 argumentos para um arranque assíncrono, e 3,4 ou 5 para arranque síncrono. Um processo iniciado por essas funções irá guardar algumas informações necessárias para um processo numa árvore de supervisão. Se o processo terminar por uma razão que seja diferente de `normal` ou `shutdown`, irá ser criado um relatório de erro. No exemplo seguinte o processo é iniciado síncrono, pela chamada de `ch4:start_link()`.

```
start_link() ->
   proc_lib:start_link(ch4, init, [self()]).
```

A função `start_link` chama a função `proc_lib:start_link`, que faz um `spawn` e liga-se ao novo processo, executando `ch4:init(Pid)`, onde Pid é o identificador do processo pai. Na função init é tratada toda a inicialização, incluindo o registo do nome do processo. O processo também tem de dar conhecimento ao seu pai que entrou em funcionamento:

```
init(Parent) ->
   ...
   proc_lib:init_ack(Parent, {ok, self()}),
   loop(...).
```

### 4.7.5. Debugging

Para usar as facilidades de *debug*, inicializa-se um termo chamado `Deb` na função `init`, através da função `sys:debug_options/1`:

```
init(Parent) ->
   ...
   Deb = sys:debug_options([]),
   ...
   loop(Chs, Parent, Deb).
```

A função `sys:debug_options/1` leva como argumento uma lista de opções. Neste caso a lista está vazia, o que significa que o *debug* está inicialmente desactivado.

Depois, para cada evento de sistema deve ser chamada a função `sys:handle_debug(Deb, Func, Info, Event)`. Eventos de sistema podem ser:

- Mensagens de entrada, representadas como `{in, Msg}` ou `{in, Msg, From}`.
- Mensagens de saída, representadas como `{out, Msg , To}`.
- Eventos de sistema definidos pelo utilizador.

No exemplo, `handle_debug` é chamado para cada mensagem, tanto de entrada como de saída. A função de formatação `Func` é a função `ch4:write_debug(Dev, Event, Info)`, que imprime as mensagens.

```
loop(Chs, Parent, Deb) ->
  receive
    {From, alloc} ->
      Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                  ch4, {in, alloc, From}),
      {Ch, Chs2} = alloc(Chs),
      From ! {ch4, Ch},
      Deb3 = sys:handle_debug(Deb2, {ch4, write_debug},
                  ch4, {out, {ch4, Ch}, From}),
      loop(Chs2, Parent, Deb3);
    {free, Ch} ->
      Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                  ch4, {in, {free, Ch}}),
      Chs2 = free(Ch, Chs),
      loop(Chs2, Parent, Deb2);
    ...
  end.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

### 4.7.6. Tratamento de Mensagens de Sistema

As mensagens de sistema são tuplos `{system, From, Request}`. O processo não necessita de interpretar o conteúdo e significado destas mensagens. Em vez disso deve chamar a função `sys:handle_system_msg(Request, From, Parent, Module, Deb, State)` que vai tratar a mensagem e em seguida chamar `Module:system_continue(Parent, Deb, State)` se a execução do processo deva continuar ou `Module:system_terminate(Reason, Parent, Deb, State)` se deva terminar.

```
loop(Chs, Parent, Deb) ->
  receive
    ...

    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent,
                  ch4, Deb, Chs)
  end.

system_continue(Parent, Deb, Chs) ->
  loop(Chs, Parent, Deb).

system_terminate(Reason, Parent, Deb, Chs) ->
```

```
exit(Reason).
```

Se o processo estiver implementado para apanhar os sinais de exit e o processo pai terminar, o comportamento esperado é que o processo filho termine pela mesma razão:

```
init(...) ->
   ...,
   process_flag(trap_exit, true),
   ...,
   loop(...).

loop(...) ->
   receive
      ...

      {exit, Parent, Reason} ->
         ..maybe some cleaning up here..
         exit(Reason);
      ...
   end.
```

## 4.8. Aplicações

Quando temos código que implementa uma funcionalidade específica, podemos tornar esse código numa aplicação, ou seja, um componente que pode ser usado noutras ocasiões.

Para fazer isso criamos um módulo de *callback* de aplicações, onde é descrito como é que a aplicação deve ser iniciada e terminada. Precisamos ainda de uma especificação da aplicação, que é guardada num ficheiro de recursos da aplicação. Entre outras coisas, são especificados que módulos fazem parte da aplicação e o nome do módulo de *callback*.

### 4.8.1. O Módulo de Callback de Aplicações

A forma como se inicia e termina a execução do código para a aplicação em causa, é descrito por duas funções de *callback*, para uma árvore de supervisão:

- `start(StartType, StartArgs) -> {ok, Pid} | {ok, Pid, State}`
- `stop(State)`

A função `start` é invocada quando se inicia a aplicação, e deve criar a árvore de supervisão, iniciando o seu supervisor de topo, a raiz. `StartType` é normalmente o átomo normal, mas pode contar outros valores, mas só em caso de *takeover* ou *failover*. `StartArgs` é definido pela chave `mod` no ficheiro de recursos da aplicação.

A função `stop` é chamada depois da aplicação ter sido terminada, para executar algum código de limpeza que seja necessário.

Em seguida temos um exemplo de um módulo de *callback* de aplicação para empacotar a árvore de supervisão usada no capítulo sobre o comportamento Supervisor.

```
-module(ch_app).
-behaviour(application).

-export([start/2, stop/1]).

start(_Type, _Args) ->
   ch_sup:start_link().

stop(_State) ->
   ok.
```

### 4.8.2. O Ficheiro de Recursos da Aplicação

Para definir uma aplicação, criamos uma especificação da aplicação que é guardada neste ficheiro de recursos:

```
{application, Application, [Opt1,...,OptN]}.
```

- `application` é um átomo qu representa o nome da aplicação.
- `Apllication` é o nome do ficheiro.
- `OptX` são tuplos `{Key, Value}` que definem propriedades da aplicação.

O conteúdo mínimo de um ficheiro deste tipo para uma árvore de supervisão:

```
{application, ch_app, [{mod, {ch_app,[]}}]}.
```

A chave `mod` define o módulo de *callback* e o argumento de inicio da aplicação. Isto significa que a função `ch_app:start(normal, [])` vai ser

chamada quando a aplicação for iniciada e `ch_app:stop([])` depois da aplicação ter sido terminada.

### 4.8.3. Estrutura de Directórios

Quando se empacota código utilizando `systools`, o código de cada aplicação é colocado em directórios separados `lib/Application-Vsn`, onde `Vsn` representa a versão. Isto é importante porque o próprio Erlang/OTP é empacotado de acordos com os princípios do OTP, e portanto apresenta uma estrutura de directórios semelhante. O servidor de código vai usar automaticamente usar código do directório com o mais alto número de versão. O directório de aplicações apresenta a seguinte estrutura:

- src – contém o código fonte do Erlang.
- ebin – contém o código objecto do Erlang, os ficheiros `beam`, bem como os recursos da aplicação.
- priv – usado para ficheiros específicos da aplicação.
- include – usada para incluir ficheiros.

### 4.8.4. Controlador de Aplicações

Quando o sistema de *run-time* do Erlang arranca, é iniciado um conjunto de processos que fazem parte da aplicação **Kernel**. Um desses processos é o controlador de aplicações, registado como `application_controller`. Todas as operações sobre as aplicações são coordenadas por esse processo

### 4.8.5. Carregar e Descarregar Aplicações

Antes duma aplicação poder ser iniciada, ela tem de ser carregada. O controlador de aplicações lê e escreve informações do ficheiro de recursos da aplicação.

```
1> application:load(ch_app).
ok
2> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.8.1.3"},
 {stdlib,"ERTS  CXC 138 10","1.11.4.3"},
 {ch_app,"Channel allocator","1"}]
```

### 4.8.6. Iniciar e Terminar Aplicações

Uma aplicação é iniciada chamando da seguinte forma:

```
5> application:start(ch_app).
ok
6> application:which_applications().
[{kernel,"ERTS  CXC 138 10","2.8.1.3"},
 {stdlib,"ERTS  CXC 138 10","1.11.4.3"},
 {ch_app,"Channel allocator","1"}]
```

Se aplicação ainda não estiver carregada, o controlador irá primeiro carregá-la. Depois vai verificar os valores das chaves da aplicação para se certificar que todas as aplicações necessárias para executar esta já estão em execução.

Em seguida é criado um *master* para a aplicação, que é um líder de todos os processos da aplicação. Este *master* inicia a aplicação chamando a função de *callback* `start/2`, com o argumento defino pela chave `mod` definida no ficheiro de recursos da aplicação.

Uma aplicação é terminada, mas não descarregada, da seguinte forma:

```
7> application:stop(ch_app).
Ok
```

O *master* termina a aplicação enviando uma mensagem ao processo supervisor do topo, a raiz, para efectuar `shutdown`. Esse processo comunica depois aos seus filhos para efectuarem `shutdown`, e a árvore é terminada pela ordem inversa de como foi iniciada. O *master* chama então a função de *callback* `stop/1` definida pela chave `mod`.

### 4.8.7. Configurar uma Aplicação

Uma aplicação pode ser configurada através dos parâmetros de configuração, que são uma lista de tuplos `{Par, Val}`, especificados pela chave `env` do ficheiro **.app**.

```
{application, ch_app,
 [{description, "Channel allocator"},
  {vsn, "1"},
  {modules, [ch_app, ch_sup, ch3]},
  {registered, [ch3]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {ch_app,[]}},
  {env, [{file, "/usr/local/log"}]}
 ]}.
```

Uma aplicação pode devolver o valor dos seus parâmetros da configuração através da chamada da função `application:get_env(App, Par)`.

```
% erl
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> application:start(ch_app).
ok
2> application:get_env(ch_app, file).
{ok,"/usr/local/log"}
```

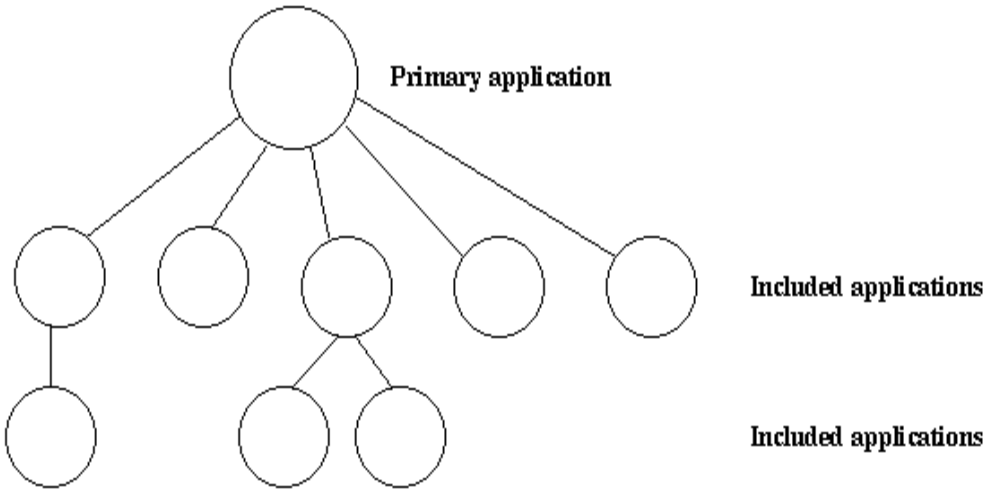
#### 4.8.8. Modos de Aplicação

Uma aplicação é iniciada em um de três modos possíveis. O modo é definido quando se inicia a aplicação, usando a função `application:start(Application, Mode)`, e especifica o que acontece quando a aplicação termina. Os modos podem ser `temporary`, `permanent` ou `transient`.

- Se uma aplicação `permanent` termina, todas as outras aplicações e o sistema de *run-time* terminam também.
- Se uma aplicação `transient` termina por uma razão normal, é gerado um relatório. Se terminar de forma anormal, todas as outras aplicações e o sistema de *run-time* são terminados também.
- Se uma aplicação `temporary` termina é gerado um relatório.

## 4.9. Aplicações Incluídas

Uma aplicação pode incluir outras aplicações. Uma aplicação incluída tem o seu próprio directório e ficheiro de recursos, mas é iniciada como parte da árvore de supervisão de outra aplicação. Uma aplicação só pode ser incluída por uma outra aplicação. Uma aplicação incluída pode incluir outra aplicação. Uma aplicação que não seja incluída por nenhuma outra é chamada de aplicação primária.




• Figura 6 – Aplicações Primárias e Incluídas

O controlador de aplicações vai carregar automaticamente uma aplicação incluída quando estiver a carregar uma aplicação primária, mas não a vai iniciar. O supervisor de topo da aplicação incluída terá de ser iniciado por um supervisor da aplicação que a inclui.

### 4.9.1. Especificação de Aplicações Incluídas

As aplicações a serem incluídas são definidas pela chave `included_applications` do ficheiro **.app**.

```
{application, treeapp,
 [{description, "Tree application"},
  {vsn, "1"},
  {modules, [treeapp_cb, treeapp_sup, treeapp_server]},
  {registered, [treeapp_server]},
```

```
 {included_applications, [subtreeapp]},
 {applications, [kernel, stdlib, sasl]},
 {mod, {treeapp_cb,[]}},
 {env, [{file, "/usr/local/log"}]}
]}.
```

### 4.9.2. Sincronizar Processos durante o Arranque

A árvore de supervisão de uma aplicação incluída é iniciada como sendo parte dar árvore de supervisão da aplicação que a inclui. Se houver necessidades de sincronização entre processos, isso pode ser conseguido através de fases de inicio, que são definidas na chave `start_phases` no ficheiro **.app**.

```
{application, treeapp,
 [{description, "Tree application"},
  {vsn, "1"},
  {modules, [treeapp_cb, treeapp_sup, treeapp_server]},
  {registered, [treeapp_server]},
  {included_applications, [subtreeapp]},
  {start_phases, [{init,[]}, {go,[]}]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {treeapp_cb,[]}},
  {env, [{file, "/usr/local/log"}]}
 ]}.

{application, subtreeapp,
 [{description, "Included application"},
  {vsn, "1"},
  {modules, [subtreeapp_sup, subtreeapp_server]},
  {registered, []},
  {start_phases, [{go,[]}]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {subtreeapp_cb,[]}}
 ]}.
```

Uma fase de início é definida por um tuplo `{Phase, PhaseArgs}`. Quando se inicia uma aplicação primária, esta inicia da maneira habitual. O controlador de aplicações cria um *master* que chama a função `Module:start(normal, StartArgs)`.

Para a aplicação primária e para cada aplicação incluída, seguindo uma abordagem *top-down*, da esquerda para a direita, o *master* chama a função a

função `Module:start_phase(Phase, Type, PhaseArgs)` para cada fase definida pela aplicação primária.

## 4.10. Aplicações Distribuídas

Num sistema distribuído com vários nós Erlang pode haver necessidade de controlar aplicações de forma distribuída. Se o nó onde uma certa aplicação estiver a correr tiver um problema, a aplicação deve ser reiniciada em outro nó.

Uma aplicação com estas características é chamada de aplicação distribuída. Mas há que notar que é o controlo da aplicação que é distribuído. Como uma aplicação distribuída pode ser executada em vários nós, há necessidade de haver um mecanismo de endereçamento para possibilitar que outras aplicações lhe possam aceder, independentemente do nó em que esta esteja a ser executada.

### 4.10.1. Especificação de Aplicações Distribuídas

As aplicações distribuídas são controladas pelo processo controlador de aplicações e pelo processo controlador de aplicações distribuídas chamado `dist_ac`. Ambos os processos fazem parte da aplicação **Kernel**, e por isso as aplicações distribuídas são especificadas através da configuração do Kernel, através dos parâmetros de configuração seguintes:

```
distributed = [{Application, [Timeout,] NodeDesc}]
```

Isto especifica onde a aplicação de nome `Application` pode executar. Os nós onde ela pode executar são dados pelo parâmetro `NodeDesc` que é uma lista de nós. `Timeout` é um inteiro que especifica quantos milissegundos se deve esperar para reiniciar a aplicação em outro nó.

Para que a distribuição do controlo de aplicações funcione correctamente, os nós onde uma aplicação pode correr têm de se contactar uns aos outros e negociar onde iniciar a aplicação. Isto é feito usando os seguintes parâmetros de configuração do **Kernel**:

- `sync_nodes_mandatory = [Node]` – especifica que outros nós têm de ser reiniciados.

- `Sync_nodes_optional = [Node]` – especifica que outros nós podem ser reiniciados.
- `Sync_nodes_timeout = integer() | infinity` – especifica quantos milissegundos esperar pelo inicio dos outros nós.

Quando é iniciado, um nó vai esperar que os outros estejam operacionais. Quando todos estiverem operacionais ou quando tiver esgotado o tempo de timeout, as aplicações começam a ser iniciadas. Se quando ocorre o timeout ainda houver nós mandatários por iniciar, o nó termina.

```
[{kernel,
  [{distributed, [{myapp, 5000, [cp1@cave, {cp2@cave, cp3@cave}]}]},
   {sync_nodes_mandatory, [cp2@cave, cp3@cave]},
   {sync_nodes_timeout, 5000}
  ]
 }
].
```

### 4.10.2. Iniciar Aplicações Distribuídas

Quando todos os nós mandatários tiverem sido iniciados, a aplicação distribuída pode ser iniciada através da chamada da função `application:start(Application)` **em todos esses nós**.

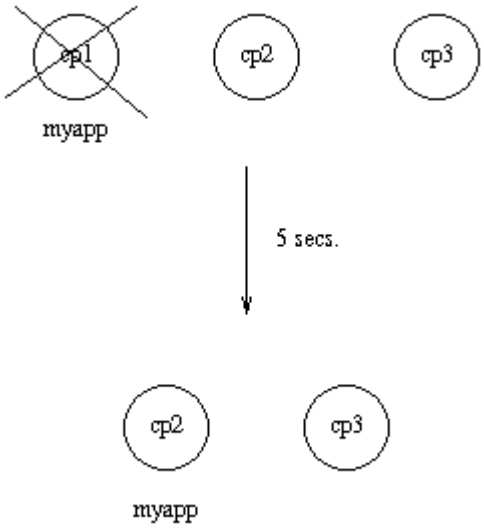
A aplicação será iniciada no primeiro nó, especificado pelos parâmetros de configuração. A aplicação é iniciada da forma habitual, com a criação de um *master* que chama a função de *callback* `Module:start(normal, StartArgs).`

### 4.10.3. Failover

Se um nó onde uma aplicação estiver a ser executada tiver um problema, a aplicação é reiniciada no primeiro nó especificado pelos parâmetros de especificação. Isto é chamado de ***Failover***.

A aplicação é iniciada de forma normal, quando o *master* chama a função `Module:start(normal, StartArgs)`, a não ser que a aplicação tenha fases de iniciação, pelo que nesse caso terá de ser iniciada recorrendo a `Module:start({failover, Node}, StartArgs).`

De acordo com o código exemplo anterior, se o nó `cp1` tiver um problema, o sistema verifica qual dos nós restantes tem menos aplicações em execução, mas espera 5 segundos para que o nó `cp1` reinicie. Se esse nó não reiniciar e `cp2` for o nó com menos aplicações em execução, a aplicação `myapp` é reiniciada nele.

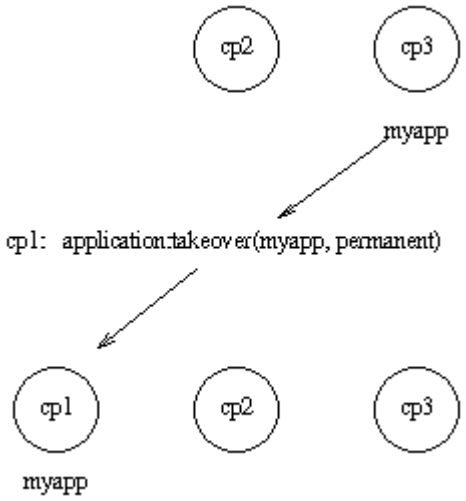



• Figura 7 - Failover

#### 4.10.4. Takeover

Se um nó é iniciado e tiver uma prioridade mais alta que o nó onde a aplicação está actualmente a ser executada, de acordo com os parâmetros de especificação, a aplicação será terminada nesse nó e reiniciada no novo nó. Isso é feito através da função `Module:start({takeover, Node}, StartArgs)`.

Se o nó `cp1` reiniciar, a função `application:takeover/2` move a aplicação `myapp` para o nó `cp1` porque este tem uma prioridade maior que `cp3` para esta aplicação.




• Figura 8 - Takeover

## 4.11. Releases

Quando temos uma ou mais aplicações pode haver a necessidade de constituir um sistema com essas aplicações e um conjunto de aplicações da plataforma Erlang/OTP. A isto chama-se uma *release*, e é feito criando-se um ficheiro de recursos da *release*. Este ficheiro define quais são as aplicações incluídas na *release*.

Este ficheiro é usado para gerar scripts de boot e pacotes de *release*. A um sistema que é transferido e instalado em outro site é chamado de **target system**. Num capítulo seguinte (Princípios de Sistema) será explicado como usar os pacotes de *releases* para criar ***target systems***.

### 4.11.1. Ficheiro de Recursos da Release

Este ficheiro define uma *release* e tem a designação de ficheiro **.rel**:

```
{release, {Name,Vsn}, {erts, EVsn},
 [{Application1, AppVsn1},
  ...
 {ApplicationN, AppVsnN}]}.
```

- O ficheiro tem de tomar o nome `Rel.rel`, onde `Rel` é um nome unívoco.
- `Name` e `Vsn` são o nome e versão da *release*.
- `Evsn` é a versão do sistema de *run-time* do Erlang ao qual a *release* se destina.

Temos agora um exemplo da criação de um *release* para a aplicação `ch_app` exemplo usada no capítulo sobre **Aplicações**:

```
{application, ch_app,
 [{description, "Channel allocator"},
  {vsn, "1"},
  {modules, [ch_app, ch_sup, ch3]},
  {registered, [ch3]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {ch_app,[]}}
 ]}.
```

O ficheiro de recursos tem de contem também as aplicações **Kernel**, **stdLib** e **SASL**, uma vez que são necessárias pela aplicação `ch_app`:

```
{release,
 {"ch_rel", "1"},
 {erts, "5.2.3.6"},
 [{kernel, "2.8.1.3"},
  {stdlib, "1.11.4.3"},
  {sasl, "1.9.4"},
  {ch_app, "1"}]
}.
```

### 4.11.2. Gerar Scripts de Boot

Estão disponíveis ferramentas no módulo `systools` para a construção e verificação de *releases*. Essas funções lêem os ficheiros de recursos da *release* e da aplicação e executam uma verificação de sintaxe e dependências. É usada a função `systools:make_script` para gerar o script de boot.

```
1> systools:make_script("ch_rel-1", [local]).
```

```
ok
```

Esta instrução cria o script `ch_rel-1.script` o parâmetro `local` é uma opção que indica que os directórios onde as aplicações estão. É depois necessário gerar uma versão binária deste script:

```
2> systools:script2boot("ch_rel-1").
ok
```

Quando se inicia o Erlang utilizando o script, todas as aplicações do ficheiro de recursos da *release* são iniciadas automaticamente:

```
% erl -boot ch_rel-1
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1>
=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
        supervisor: {local,sasl_safe_sup}
          started: [{pid,<0.33.0>},
                    {name,alarm_handler},
                    {mfa,{alarm_handler,start_link,[]}},
                    {restart_type,permanent},
                    {shutdown,2000},
                    {child_type,worker}]

...

=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
        application: sasl
         started_at: nonode@nohost

...
=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
        application: ch_app
         started_at: nonode@nohost
```

### 4.11.3. Criar um Pacote de Releases

Usa-se a função `systools:make_tar` que pega num ficheiro `.rel` e cria um ficheiro compactado **.tar** com o código das respectivas aplicações, que é chamado de pacote da *release*.

```
1> systools:make_script("ch_rel-1").
ok
2> systools:script2boot("ch_rel-1").
ok
3> systools:make_tar("ch_rel-1").
ok
```

Por defeito, o pacote contém os ficheiros `.app` e o código objecto de todas as aplicações, estruturado de acordo com a Estrutura de Directórios da Aplicação. Contém ainda o script de boot binário renomeado `start.boot` e o ficheiro de recursos da *release*, o ficheiro `.rel`.

```
% tar tf ch_rel-1.tar
lib/kernel-2.8.1.3/ebin/kernel.app
lib/kernel-2.8.1.3/ebin/application.beam
...
lib/stdlib-1.11.4.3/ebin/stdlib.app
lib/stdlib-1.11.4.3/ebin/beam_lib.beam
...
lib/sasl-1.9.4/ebin/sasl.app
lib/sasl-1.9.4/ebin/sasl.beam
...
lib/ch_app-1/ebin/ch_app.app
lib/ch_app-1/ebin/ch_app.beam
lib/ch_app-1/ebin/ch_sup.beam
lib/ch_app-1/ebin/ch3.beam
releases/1/start.boot
releases/ch_rel-1.rel
```

Podem ser definidas opções para que o pacote inclua também o código fonte e o binário do sistema de *run-time* do Erlang. Os pacotes devem ser descompactados e instalados usando o *handler* da *release*.

### 4.11.4. Estrutura de Directórios

A estrutura de directórios para o código instalado pelo *handler* da *release* é o seguinte:

```
$ROOTDIR/lib/App1-AVsn1/ebin
                /priv
        /App2-AVsn2/ebin
                /priv
        ...
        /AppN-AVsnN/ebin
```

```
        /priv
/erts-EVsn/bin
/releases/Vsn
/bin
```

- `lib` – Directório das aplicações. O ficheiro de script boot deve estar localizado no directório `releases/Vsn`, em que `Vsn` é a versão da *release*.
- `erts-EVsn/bin` – contém os ficheiros executáveis do sistema de *run-time*.
- `releases/Vsn` – contém os ficheiros `.rel`.
- `bin` – contém o ficheiro executável **erl**.

# 5. Princípios de Sistema

## 5.1. Arrancar o Sistema

O sistema de *run-time* do Erlang é arrancado através do comando erl:

```
Erlang (BEAM) emulator version 5.2.3.5 [hipe] [threads:0]
Eshell V5.2.3.5  (abort with ^G)
1>
```

Esta aplicação entende vários argumentos da linha de comandos, e as aplicações podem aos valores dos parâmetros da linha de comandos através da invocação da função `init:get_argument(Key)`, ou `init:get_arguments()`.

## 5.2. Iniciar e Terminar o Sistema

O Sistema pode ser interrompido através do uso da função `halt`, com 1 ou nenhum argumento. As funções para iniciar, reiniciar e terminar o sistema encontram-se no módulo `init`:

- `init:restart().`
- `init:reboot().`
- `init:stop().`

O sistema de *run-time* é também terminado se a *shell* do Erlang for fechada.

## 5.3. Scripts de Boot

### 5.3.1. Script de Boot Default

O sistema de *run-time* é iniciado através de um script de boot. Este script contém instruções sobre qual o código que deve ser carregado e quais as aplicações e processos que devem ser iniciados. Um ficheiro de script de boot tem a extensão **.script**. o sistema de *run-time* usa uma versão binária deste script, um ficheiro com extensão **.boot**.

A plataforma Erlang/OTP vem com dois scripts incluídos:

- `start_clean.boot` – carrega o código necessário para as aplicações **Kernel** e **stdLib** e inicia essas aplicações.
- `start_sasl.boot` – carrega o código necessário para as aplicações **Kernel**, **stdLib** e **SASL** e inicia essas aplicações.

O utilizador define qual destes scripts pretende usar quando instala o Erlang/OTP.

### 5.3.2. Scripts de Boot definidos pelo Utilizador

É possível escrever scripts de boot manualmente, mas no entanto a maneira recomendada de o fazer é através do ficheiro de recursos de uma *release*, através da função `systools:make_script`. Isto requer no entanto que o código esteja estruturado de acordo com os Princípios de Programação em OTP.

O ficheiro binário do script é gerado através do uso da função `systools:script2boot(File)`. `File` representa o ficheiro **.script**.

Para iniciar o sistema de *run-time* usando o ficheiro de boot criado:

```
% erl -boot <Name>
```

## 5.4. Estratégias para Carregar Código

O sistema de *run-time* pode ser arrancado em dois modos, **embedded** ou **interactive**. Para decidir em qual modo pretendemos arrancar, usa-se a *flag* `–mode`:

```
% erl -mode <Mode>
```

- Ø Modo **Embedded** – todo o código é carregado durante o arranque do sistema, de acordo com o script de boot.
- Ø Modo **Interactive** – o código é carregado dinamicamente à medida que vai sendo referenciado. É o modo predefinido.

## 5.5. Criar o primeiro Target System

Quando se cria um sistema Erlang/OTP, a maneira mais simples de o fazer é instalar o Erlang num sítio, instalar o código específico das aplicações noutro, e depois arrancar o sistema de *run-time*, certificando-se que o *code path* inclui o código específico das aplicações.

Mas um programador pode criar uma aplicação para um determinado fim específico, em que várias aplicações do sistema Erlang/OTP são irrelevantes para esse fim. Há então uma necessidade de criar um novo sistema baseado no Erlang/OTP, onde as aplicações desnecessárias são removidas, e onde é incluído um conjunto de aplicações necessárias.

A isto chama-se criar um Target System.

### 5.5.1. Criar um Target System

Supondo que se tem um sistema Erlang/OTP estruturado de acordo com os Princípios da Programação em OTP, seguem-se os passos seguintes. Todos os exemplos são para criar um target system em UNIX.

**Passo 1:** Criar um ficheiro de recursos de *release* que especifica a versão do sistema de *run-time* e uma lista das aplicações que devem ser incluídas neste novo target system.

```
%% mysystem.rel
{release,
 {"MYSYSTEM", "FIRST"},
 {erts, "5.1"},
 [{kernel, "2.7"},
  {stdlib, "1.10"},
  {sasl, "1.9.3"},
  {pea, "1.0"}]}.
```

As aplicações podem não ser apenas aplicações Erlang/OTP originais, mas também aplicações escritas pelo utilizador que está a criar o target system.

**Passo 2:** Iniciar o sistema Erlang/OTP a partir da localização onde se encontra o ficheiro `mysystem.rel`.

```
erl -pa /home/user/target_system/myapps/pea-1.0/ebin
```

É também fornecido o caminho do directório `ebin` da aplicação `pea`, uma vez que esta não faz parte do sistema Erlang/OTP original.

**Passo 3:** Criar o target system.

```
target_system:create("mysystem").
```

Esta função faz o seguinte:

- Lê o ficheiro de recursos da *release* e cria um novo ficheiro `plain.rel` idêntico, excepto que apenas vai listar nele as aplicações **Kernel** e **stdLib**.

- A partir desses dois ficheiros cria dois pares de ficheiros de script de boot, um de texto e o outro binário: `mysystem.script` e `.boot` e `plain.script` e `.boot`.
- Cria o pacote `mysystem.tar.gz` que inclui
  - `erts-5.1/bin/`
  - `releases/FIRST/start.boot`
  - `releases/mysystem.rel`
  - `lib/kernel-2.7/`
  - `lib/stdlib-1.10/`
  - `lib/sasl-1.9.3/`
  - `lib/pea-1.0/`
- Cria o directório temporário `tmp` e extrai para ele o conteúdo do referido pacote.
- Apaga os ficheiros `erl` e `start` do directório `tmp/erts-5.1/bin`.
- Cria o directório `tmp/bin`.
- Copia o ficheiro `plain.boot` para `tmp/bin/start.boot`.
- Copia os ficheiros `epmd`, `run_erl` e `to_erl` do directório `tmp/erts-5.1/bin` para o directório `tmp/bin`.
- Cria o ficheiro `tmp/releases/start_erl.data` com o conteúdo "`5.1 FIRST`".
- Volta a compactar o ficheiro `mysystem.tar.gz` a partir do directório `tmp` e remove-o.

### 5.5.2. Instalar um Target System

O target system é instalado através do uso da função seguinte:

```
3> target_system:install("mysystem", "/usr/local/erl-target").
```

O que esta função faz é:

- Extrai o ficheiro `mysystem.tar.gz` para `/usr/local/erl-target`.
- Lê o conteúdo do ficheiro `releases/start_erl.data` para saber a versão do sistema de *run-time*.
- Substitui, nos ficheiros `erl.src`, `start.src` e `start_erl.src` do directório `erts-5.1/bin`, `%FINAL_ROOTDIR%` e `%EMU%` por

`/usr/local/erl-target` e `beam`, respectivamente e coloca os ficheiros resultantes no directório `bin`.

- É criado o ficheiro `releases/RELEASES` com dados constantes do ficheiro `releases/mysystem.rel`.

## 5.6. Listagem do ficheiro target_system.erl

```
-module(target_system).
-include_lib("kernel/include/file.hrl").
-export([create/1, install/2]).
-define(BUFSIZE, 8192).

%% Note: RelFileName below is the *stem* without trailing .rel,
%% .script etc.
%%

%% create(RelFileName)
%%
create(RelFileName) ->
    RelFile = RelFileName ++ ".rel",
    io:fwrite("Reading file: \"~s\" ...~n", [RelFile]),
    {ok, [RelSpec]} = file:consult(RelFile),
    io:fwrite("Creating file: \"~s\" from \"~s\" ...~n",
              ["plain.rel", RelFile]),
    {release,
     {RelName, RelVsn},
     {erts, ErtsVsn},
     AppVsns} = RelSpec,
    PlainRelSpec = {release,
                    {RelName, RelVsn},
                    {erts, ErtsVsn},
                    lists:filter(fun({kernel, _}) ->
                                         true;
                                    ({stdlib, _}) ->
                                         true;
                                    (_) ->
                                         false
                                 end, AppVsns)
                   },
    {ok, Fd} = file:open("plain.rel", [write]),
    io:fwrite(Fd, "~p.~n", [PlainRelSpec]),
    file:close(Fd),

    io:fwrite("Making \"plain.script\" and \"plain.boot\" files ...~n"),
    make_script("plain"),

    io:fwrite("Making \"~s.script\" and \"~s.boot\" files ...~n",
              [RelFileName, RelFileName]),
    make_script(RelFileName),

    TarFileName = io_lib:fwrite("~s.tar.gz", [RelFileName]),
```

```
    io:fwrite("Creating tar file \"~s\" ...~n", [TarFileName]),
    make_tar(RelFileName),

    io:fwrite("Creating directory \"tmp\" ...~n"),
    file:make_dir("tmp"),

    io:fwrite("Extracting \"~s\" into directory \"tmp\" ...~n", [TarFileName]),
    extract_tar(TarFileName, "tmp"),

    TmpBinDir = filename:join(["tmp", "bin"]),
    ErtsBinDir = filename:join(["tmp", "erts-" ++ ErtsVsn, "bin"]),
    io:fwrite("Deleting \"erl\" and \"start\" in directory \"~s\" ...~n",
            [ErtsBinDir]),
    file:delete(filename:join([ErtsBinDir, "erl"])),
    file:delete(filename:join([ErtsBinDir, "start"])),

    io:fwrite("Creating temporary directory \"~s\" ...~n", [TmpBinDir]),
    file:make_dir(TmpBinDir),

    io:fwrite("Copying file \"plain.boot\" to \"~s\" ...~n",
            [filename:join([TmpBinDir, "start.boot"])]),
    copy_file("plain.boot", filename:join([TmpBinDir, "start.boot"])),

    io:fwrite("Copying files \"epmd\", \"run_erl\" and \"to_erl\" from \n"
            "\"~s\" to \"~s\" ...~n",
            [ErtsBinDir, TmpBinDir]),
    copy_file(filename:join([ErtsBinDir, "epmd"]),
            filename:join([TmpBinDir, "epmd"]), [preserve]),
    copy_file(filename:join([ErtsBinDir, "run_erl"]),
            filename:join([TmpBinDir, "run_erl"]), [preserve]),
    copy_file(filename:join([ErtsBinDir, "to_erl"]),
            filename:join([TmpBinDir, "to_erl"]), [preserve]),

    StartErlDataFile = filename:join(["tmp", "releases", "start_erl.data"]),
    io:fwrite("Creating \"~s\" ...~n", [StartErlDataFile]),
    StartErlData = io_lib:fwrite("~s ~s~n", [ErtsVsn, RelVsn]),
    write_file(StartErlDataFile, StartErlData),

    io:fwrite("Recreating tar file \"~s\" from contents in directory "
            "\"tmp\" ...~n", [TarFileName]),
    {ok, Tar} = erl_tar:open(TarFileName, [write, compressed]),
    {ok, Cwd} = file:get_cwd(),
    file:set_cwd("tmp"),
    erl_tar:add(Tar, "bin", []),
    erl_tar:add(Tar, "erts-" ++ ErtsVsn, []),
    erl_tar:add(Tar, "releases", []),
    erl_tar:add(Tar, "lib", []),
    erl_tar:close(Tar),
    file:set_cwd(Cwd),
    io:fwrite("Removing directory \"tmp\" ...~n"),
    remove_dir_tree("tmp"),
    ok.


install(RelFileName, RootDir) ->
```

```
    TarFile = RelFileName ++ ".tar.gz",
    io:fwrite("Extracting ~s ...~n", [TarFile]),
    extract_tar(TarFile, RootDir),
    StartErlDataFile = filename:join([RootDir, "releases", "start_erl.data"]),
    {ok, StartErlData} = read_txt_file(StartErlDataFile),
    [ErlVsn, RelVsn| _] = string:tokens(StartErlData, " \n"),
    ErtsBinDir = filename:join([RootDir, "erts-" ++ ErlVsn, "bin"]),
    BinDir = filename:join([RootDir, "bin"]),
    io:fwrite("Substituting in erl.src, start.src and start_erl.src to\n"
              "form erl, start and start_erl ...\n"),
    subst_src_scripts(["erl", "start", "start_erl"], ErtsBinDir, BinDir,
                      [{"FINAL_ROOTDIR", RootDir}, {"EMU", "beam"}],
                      [preserve]),
    io:fwrite("Creating the RELEASES file ...\n"),
    create_RELEASES(RootDir,
                    filename:join([RootDir, "releases", RelFileName])).

%% LOCALS

%% make_script(RelFileName)
%%
make_script(RelFileName) ->
    Opts = [no_module_tests],
    systools:make_script(RelFileName, Opts).

%% make_tar(RelFileName)
%%
make_tar(RelFileName) ->
    RootDir = code:root_dir(),
    systools:make_tar(RelFileName, [{erts, RootDir}]).

%% extract_tar(TarFile, DestDir)
%%
extract_tar(TarFile, DestDir) ->
    erl_tar:extract(TarFile, [{cwd, DestDir}, compressed]).

create_RELEASES(DestDir, RelFileName) ->
    release_handler:create_RELEASES(DestDir, RelFileName ++ ".rel").

subst_src_scripts(Scripts, SrcDir, DestDir, Vars, Opts) ->
    lists:foreach(fun(Script) ->
                          subst_src_script(Script, SrcDir, DestDir,
                                           Vars, Opts)
                  end, Scripts).

subst_src_script(Script, SrcDir, DestDir, Vars, Opts) ->
    subst_file(filename:join([SrcDir, Script ++ ".src"]),
               filename:join([DestDir, Script]),
               Vars, Opts).

subst_file(Src, Dest, Vars, Opts) ->
    {ok, Conts} = read_txt_file(Src),
    NConts = subst(Conts, Vars),
    write_file(Dest, NConts),
    case lists:member(preserve, Opts) of
```

```
        true ->
            {ok, FileInfo} = file:read_file_info(Src),
            file:write_file_info(Dest, FileInfo);
        false ->
            ok
    end.

%% subst(Str, Vars)
%% Vars = [{Var, Val}]
%% Var = Val = string()
%% Substitute all occurrences of %Var% for Val in Str, using the list
%% of variables in Vars.
%%
subst(Str, Vars) ->
    subst(Str, Vars, []).

subst([$%, C| Rest], Vars, Result) when $A =< C, C =< $Z ->
    subst_var([C| Rest], Vars, Result, []);
subst([$%, C| Rest], Vars, Result) when $a =< C, C =< $z ->
    subst_var([C| Rest], Vars, Result, []);
subst([$%, C| Rest], Vars, Result) when  C == $_ ->
    subst_var([C| Rest], Vars, Result, []);
subst([C| Rest], Vars, Result) ->
    subst(Rest, Vars, [C| Result]);
subst([], _Vars, Result) ->
    lists:reverse(Result).

subst_var([$%| Rest], Vars, Result, VarAcc) ->
    Key = lists:reverse(VarAcc),
    case lists:keysearch(Key, 1, Vars) of
        {value, {Key, Value}} ->
            subst(Rest, Vars, lists:reverse(Value, Result));
        false ->
            subst(Rest, Vars, [$%| VarAcc ++ [$%| Result]])
    end;
subst_var([C| Rest], Vars, Result, VarAcc) ->
    subst_var(Rest, Vars, Result, [C| VarAcc]);
subst_var([], Vars, Result, VarAcc) ->
    subst([], Vars, [VarAcc ++ [$%| Result]]).

copy_file(Src, Dest) ->
    copy_file(Src, Dest, []).

copy_file(Src, Dest, Opts) ->
    {ok, InFd} = file:rawopen(Src, {binary, read}),
    {ok, OutFd} = file:rawopen(Dest, {binary, write}),
    do_copy_file(InFd, OutFd),
    file:close(InFd),
    file:close(OutFd),
    case lists:member(preserve, Opts) of
        true ->
            {ok, FileInfo} = file:read_file_info(Src),
            file:write_file_info(Dest, FileInfo);
        false ->
            ok
```

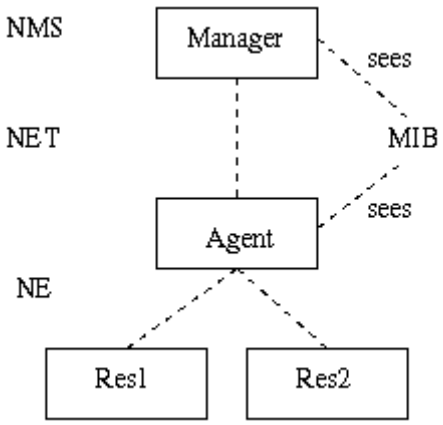
```
    end.

do_copy_file(InFd, OutFd) ->
    case file:read(InFd, ?BUFSIZE) of
        {ok, Bin} ->
            file:write(OutFd, Bin),
            do_copy_file(InFd, OutFd);
        eof  ->
            ok
    end.

write_file(FName, Conts) ->
    {ok, Fd} = file:open(FName, [write]),
    file:write(Fd, Conts),
    file:close(Fd).

read_txt_file(File) ->
    {ok, Bin} = file:read_file(File),
    {ok, binary_to_list(Bin)}.

remove_dir_tree(Dir) ->
    remove_all_files(".", [Dir]).

remove_all_files(Dir, Files) ->
    lists:foreach(fun(File) ->
                        FilePath = filename:join([Dir, File]),
                        {ok, FileInfo} = file:read_file_info(FilePath),
                        case FileInfo#file_info.type of
                            directory ->
                                {ok, DirFiles} = file:list_dir(FilePath),
                                remove_all_files(FilePath, DirFiles),
                                file:del_dir(FilePath);
                            _ ->
                                file:delete(FilePath)
                        end
                end, Files).
```

# 6. Princípios OAM

O suporte para operação e manutenção no OTP consiste num modelo genérico para manutenção de subsistemas no OTP, e em alguns componentes usados nesses subsistemas. A ideia principal deste modelo é que ele seja independente do protocolo de manutenção usado, pois não está "preso" a qualquer protocolo. É definida uma API que pode ser usada para escrever adaptações para protocolos de manutenção específicos.

Cada componente OAM é implementado como uma sub aplicação, que pode ser incluída numa aplicação de manutenção do sistema.

O modelo independente do protocolo ao nível da rede é o *Manager – Agent Model*. Este modelo é baseado no princípio cliente – servidor, onde o *manager* (cliente) envia pedidos ao agente (servidor) e o agente responde a esses pedidos. Há duas diferenças principais entre este modelo e o modelo cliente – servidor típico. Em primeiro lugar é comum haver poucos *manager*s a comunicar com muitos agentes. E em segundo, o agente pode enviar notificações espontaneamente ao *manager*.




• Figura 9 – Termino*log*ia

## 6.1. Terminologia OAM

O *manager* é muitas vezes referenciado como NMS, ou seja, *Network Management Station*, o lugar onde um operador gere a rede. O agente é uma entidade que executa dentro de um NE, ou seja, *Network Element*. No OTP, o NE pode ser um sistema distribuído, o que quer dizer que esse sistema é gerido como uma única entidade. Mas também podemos ter que o agente seja configurado para poder ser executado em vários nós, fazendo dele uma aplicação OTP distribuída.

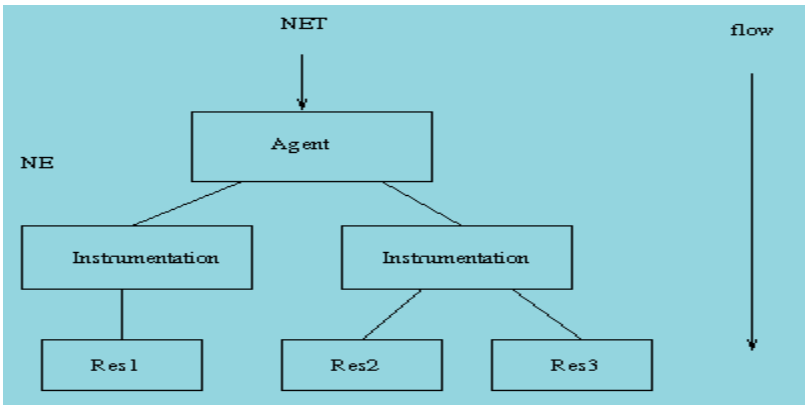
A informação de manutenção é definida num *Management Information Base* (MIB). É uma definição formal sobre quais as informações que o agente disponibiliza ao *manager*, que acede ao MIB através de um protocolo de manutenção, como o SNMP, CMIP, HTTP ou CORBA. Cada um destes protocolos têm a sua própria linguagem de definição de MIBs, o ASN.1, GDMO, está implícito e IDL, respectivamente. Normalmente as entidades definidas num MIB são chamadas *Managed Object* (MO), apesar destes objectos não terem obrigatoriamente de ser objectos como nas linguagens OO, pois tratam-se de objectos lógicos.

## 6.2. Modelo

Este modelo é usado por todos os componentes de operação e manutenção, e também pode ser usado pelas aplicações. A grande vantagem

deste modelo é que separa claramente os recursos dos protocolos de manutenção, pois os recursos não precisam de saber qual o protocolo usado para gerir o sistema. Assim é possível gerir os mesmos recursos com vários protocolos.

As diferentes entidades envolvidas neste modelo são o agente que termina o protocolo de manutenção, e o recurso que está para ser gerido. De uma maneira geral os recursos não devem ter conhecimento do protocolo usado e o agente não deve ter conhecimento sobre o recurso gerido. Isto implica que haja um mecanismo de tradução para traduzir as operações de manutenção para operações no recurso. Isto é chamado de **instrumentação**, e é implementado por uma função de instrumentação. Uma função é escrita para cada combinação de protocolo/recurso.

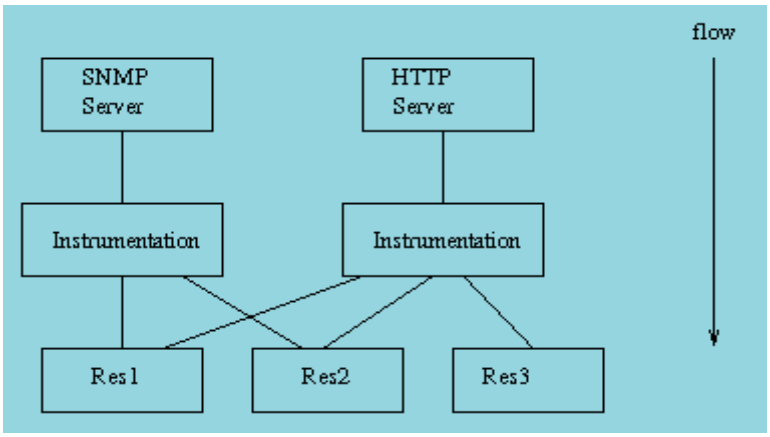



• Figura 10 – Pedido de um *manager* a um agente

O mapeamento entre funções e recursos não tem obrigatoriamente de ser de 1 para 1. É possível escrever uma função para cada recurso e usar essa função com diferentes protocolos.

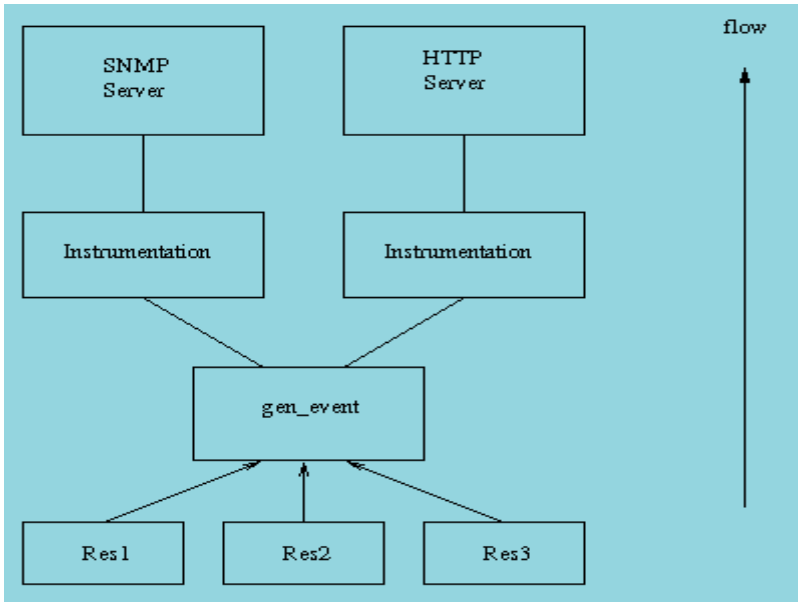
O agente recebe um pedido e mapeia essa pedido para chamar uma ou mais funções de instrumentação. As funções executam operações sobre os recursos para implementar a semântica associada com os MOs.

Temos em seguida um exemplo de um sistema gerido através de SNMP e HTTP:




• Figura 11 – Estrutura sistema gerido através de SNMP e HTTP

Os recursos também podem enviar notificações para o *manager*:




• Figura 12 – Tratamento de Notificações

A ideia principal é que os recursos enviam as notificações sob a forma de termos Erlang a um processo gen_event dedicado. Neste processo são instalados *handlers* para diferentes protocolos de manutenção. Quando um evento é recebido por este processo, é encaminhado para cada um dos *handlers* instalados, que são responsáveis por traduzir o evento numa notificação que possa ser enviada através do protocolo de manutenção.

# 7. Componentes OTP

A plataforma Erlang/OTP é constituída por um conjunto de aplicações que podem ser agrupadas em seis grupos principais. Neste capítulo são apresentadas todas essas aplicações, agrupadas pelos grupos a que pertencem.

## 7.1. Aplicações Básicas

### 7.1.1. Compiler

O compilador do Erlang traduz o código fonte em código objecto que é independente do CPU em uso, o que permite que esse código resultante possa ser usado em diversas plataformas que suportem Erlang.

### 7.1.2. ERTS

É o sistema de *run-time* do Erlang.

### 7.1.3. Kernel

É sempre a primeira aplicação a ser iniciada. Num sistema OTP mínimo, o Kernel é uma das duas únicas aplicações, juntamente com a **StdLib**. O Kernel contém os seguintes serviços:

- application_controller
- auth
- code
- error_logger
- file
- global_name_server
- net_kernel
- rpc
- user

### 7.1.4. SASL

As aplicações SASL oferecem suporte para *Log* de erros, Tratamento de alarmes, Regulação de *overload*, *release handling* e *report browsing*. Esta aplicação introduz três tipos de relatórios:

- Relatórios de supervisor
- Relatórios de progresso
- Relatórios de erros

Quando a aplicação SASL é executada, é adicionado um *handler* que formata e escreve estes relatórios, como tiver sido especificado nos parâmetros de configuração do SASL.

**Relatórios de Supervisor**

Este tipo de relatório é tratado quando um processo filho, que está a ser supervisionado, termina de maneira inesperada. Estes relatórios contêm os seguintes aspectos:

- Supervisor – o nome do supervisor do processo.

- Ø Contexto – indica em que fase o processo filho foi terminado, sob o ponto de vista do supervisor.
- Ø Razão – o motivo pelo qual o processo terminou.
- Ø Offender – especificações iniciais do processo filho

**Relatórios Progresso**

Este tipo de relatório é tratado sempre que um supervisor é iniciado ou reiniciado. Contem os seguintes itens:

- Ø Supervisor – o nome do supervisor do processo.
- Ø Started – as especificações iniciais do processo filho iniciado.

**Relatórios de Erros**

Processos que são criados com as funções `proc_lib:spawn` ou `proc_lib:spawn_link` são apanhados dentro de um *catch*. Um relatório de erro é tratado sempre que um destes processos termina sem razão aparente. Os seguintes itens fazem partes destes relatórios:

- Ø Crasher – informações sobre o processo que deu erro.
- Ø Neighbours – informação sobre processos que estão ligados ao processo que deu erro. São os processos que irão terminar devido ao erro do processo que deu erro.

**A Estrutura das *Releases***

Os programas Erlang estão organizados em módulos, em que cada módulo de uma *release* tem que ter um nome único. Colecções de módulos cooperantes na resolução de um problema particular estão organizadas em aplicações. Colecções de aplicações estão organizadas em *releases*.

Cada nova *release* é assemblada num novo pacote. Um pacote destes é instalado num sistema em execução através da passagem de comandos para o *release handler*, que é um processo SASL. Um sistema tem uma versão de sistema único, que é actualizado sempre que uma nova *release* é instalada.

### 7.1.5. stdLib

Contém as bibliotecas essenciais do Erlang para a execução do sistema. A outra aplicação obrigatória é o ***kernel***.

## 7.2. Aplicações de Base de Dados

### 7.2.1. Mnemosyne

A Mnemosyne é uma linguagem de *query* do DBMS Mnesia, e dispõem de uma sintaxe simples para *queries* complexas. As *queries* são usadas para se aceder aos dados guardados no DBMS. Uma *query* especifica uma relação entre todos os dados seleccionados. A *query* retorna um handle que é usado como argumento para diversas avaliações de funções. Depois de obtido o handle, a *query* pode ser avaliada de três maneiras diferentes:

- Uma *query* que retorna todos os resultados possíveis, através da função `eval/1`.
- Obter as respostas em "pedaços" de tamanho variável, chamados de cursores. Assim a *query* pode ser abortada quando se obtêm resultados suficientes.
- E por ultimo, um tipo de cursor ainda mais sofisticado, onde o tempo gasto na criação do cursor pode ser feito adiantadamente.

### 7.2.2. Mnesia

O Mnesia é um DBMS (*Database Management System*) distribuído, apropriado para aplicações de telecomunicações e para outras aplicações Erlang que necessitem de operações contínuas e propriedades de tempo real.

A origem do Mnesia foi provocada pela necessidade de criar um DBMS distribuído que fosse tolerante a falhas e que pudesse ser executado no mesmo espaço de endereçamento da aplicação.

Foi implementado, e está intimamente ligado à linguagem Erlang, e por isso dispõe das funcionalidades necessárias para a implementação de sistemas de telecomunicações tolerantes a falhas. O Mnesia tenta endereçar todos os aspectos relacionados com a gestão dos dados requeridos para sistemas de telecomunicações, e por isso dispõe de um conjunto de funcionalidades que não estão normalmente presentes nos DMBS tradicionais:

- Pesquisa rápida e em tempo real de chaves ou valores.
- *Queries* complicadas para manutenção podem não ser executadas em tempo real.

- Ø Dados distribuídos para aplicações distribuídas.
- Ø Alta tolerância a falhas.
- Ø Possibilidade de configuração dinâmica.
- Ø Objectos complexos.

### 7.2.3. Mnesia_session

Não é mais que uma interface para o DBMS Mnesia. Permite o acesso ao Mnesia mesmo que vindo de linguagens de programação que não o Erlang. Esta interface está definida em IDL (*Interface Definition Language*). O acesso é definido recorrendo a dois protocolos:

- Ø IIOP (*Internet Inter ORB Protocol*), que é um standard da OMG.
- Ø O protocolo proprietário da distribuição do Erlang Erl_Interface

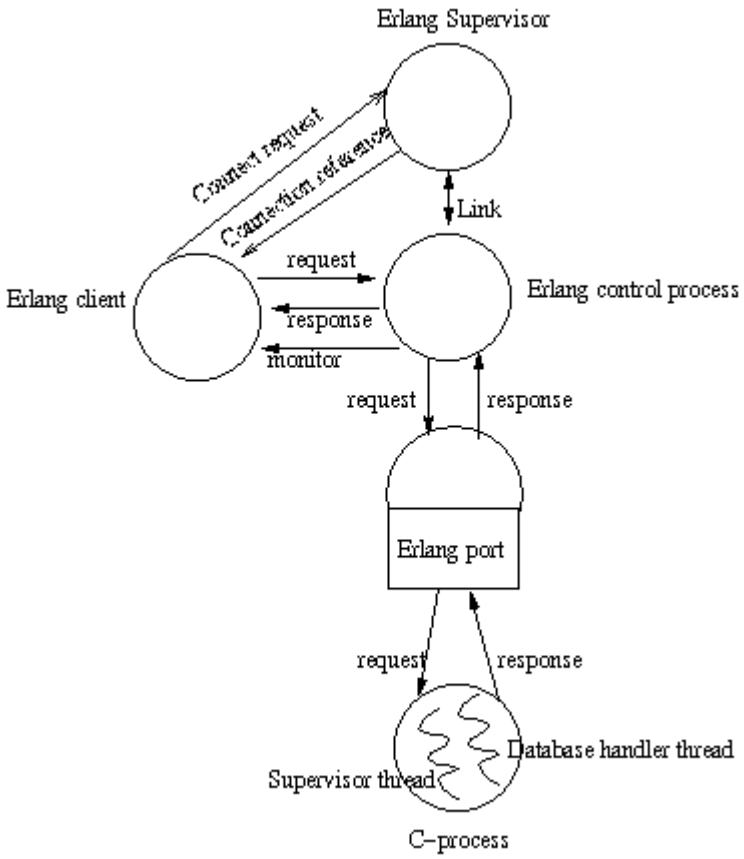
**Interfaces**

- Ø ***connector*** é iniciado de forma estática quando se inicia a aplicação
- Ø **session** é iniciado a pedido de um cliente.

Quando um cliente necessita de aceder ao Mnesia, tenta localizar um ***connector*** num qualquer nó Erlang e inicia sessão nesse nó aplicando a função *connect* no ***connector***. Depois de ter iniciado sessão, pode executar diversas operações.

### 7.2.4. ODBC

O objectivo desta aplicação é proporcionar aos programadores uma interface ODBC que esteja mais ligada ao Erlang, para que o programador necessite apenas de se concentrar na resolução do problema específico em vez de se debater com os apontadores e alocações de memória característicos de outras interfaces ODBC. Também dispõe de uma interface para comunicar com Bases de Dados Relacionais SQL. Foi desenvolvido tendo por base a interface ODBC da Microsoft e como tal, requer que se tenha um *driver* ODBC instalado para a Base de Dados a que se quer aceder.




• Figura 13 – Visão Geral do funcionamento do ODBC

Quando a aplicação ODBC é iniciada, a única coisa que acontece é a criação de um processo supervisor. Para cada chamada da função `connect/2`, é criado um novo processo que é adicionado como processo filho do supervisor ODBC do Erlang. As únicas funções do supervisor são providenciar relatórios de erros se um processo terminar de forma anormal, e possibilitar a alteração de código. Apenas o cliente tem o poder de decidir se a conexão deve ser reiniciada.

Quando é feito o *connect*, abre-se uma porta para um processo-C que trata da comunicação com a Base de Dados, através do *driver* ODBC da Microsoft. O processo-C consiste em duas *threads*, a supervisora que verifica a existência de novas mensagens na porta do Erlang, e o *handler* de Base de Dados que comunica com a Base de Dados. O efeito prático disto é que o processo detecta se o Erlang fechar a porta se por um qualquer motivo a

*thread* do *handler* de Base de Dados der erro. Neste caso o processo-C vai terminar.

## 7.3. Aplicações de Execução e de Manutenção

### 7.3.1. Eva

EVA (*EVent Alarm Handling*) é uma aplicação de Gestão de Falhas. Consiste num suporte para manusear eventos e alarme, bem como para controlo de *Logs*. Trata-se de uma aplicação modular que consiste em dois serviços de gestão de protocolos. Contêm regras e funções que permitem definir adaptações de protocolos. Uma dessas adaptações é incluída para SNMP, os MIBs do SNMP é as suas implementações. É de referir que esta aplicação usa a Mnesia e SASL.

EVA é uma sub aplicação que pode ser incorporada em outra aplicação. Está desenhada de modo a funcionar como uma aplicação distribuída, e por isso é sempre executada num nó, enquanto outros nós estão em *standby*. Para minimizar o tráfego na rede, esta aplicação deverá correr no mesmo nó onde correm as outras aplicações de manutenção.

Da maneira como foi desenhada, a EVA é independente do protocolo e por isso pode ser usada com diversos protocolos de gestão. Para cada protocolo em que é usada, tem que se escrever uma das adaptações referidas acima.

### 7.3.2. Os_mon

Esta ferramenta é usada para tarefas de controlo e monitorização de recursos do sistema operativo, como sejam a utilização do CPU e da memória.

### 7.3.3. Otp_mibs

O objectivo desta aplicação é fornecer uma base de informações de gestão de SNMP para a nós Erlang/OTP, sem fazer depender essa plataforma do SNMP. Os MIBs estão definidos na sintaxe SNMPv2 SMI.

**OTP-MIB**

Este MIB representa informação sobre os nós Erlang/OTP. Informação comum como o nome do nó, o numero de processos em execução, a versão

da máquina virtual. Se for usado num sistema, o MIB deve ser carregado num agente SNMP, usando as funções específicas disponibilizadas.

**OTP-REG**

Este módulo define a sub árvore de identificadores de objectos do OTP debaixo da sub árvore da Ericsson. Debaixo da sub árvore do OTP são definidos vários identificadores de objectos. Esta aplicação é normalmente usada por aplicações OTP que definem os seus próprios MIBs, ou pelos módulos ASN.1 que requerem um identificador de objecto único.

**OTP-TC**

Este MIB fornece a convenção textual do tipo de dados `OwnerString`. Este tipo de dados é usado, por exemplo, por MIBs na aplicação EVA para designar o proprietário das entradas de *log*.

### 7.3.4. SNMP

Esta ferramenta de desenvolvimento fornece um ambiente para uma prototipagem e construção rápidas de agentes. Esta ferramenta é usada para criar um agente multi-lingual SNMP recorrendo à seguinte informação:

- Uma descrição de um MIB em ASN.1 (*Abstract Syntax Notation One*)
- Funções de instrumentação para os objectos geridos do MIB, escritas em Erlang.

A vantagem de usar um *toolkit* de agentes extensível é a possibilidade de se remover detalhes como verificação de tipos, direitos de acesso, *Protocol Data Unit* (PDU), codificação e descodificação das mãos do programador, que assim apenas se tem de preocupar em escrever as funções de instrumentação que implementam os MIBs.

Este *toolkit* é constituído por:

- Um agente SNMP extensível que entende SNMPv1, SNMPv2, SNMPv3, ou qualquer combinação destes protocolos.
- Um compilador MIB que entende SMIv1 e SMIv2.
- Um gestor SNMP que pode ser usado para testes.

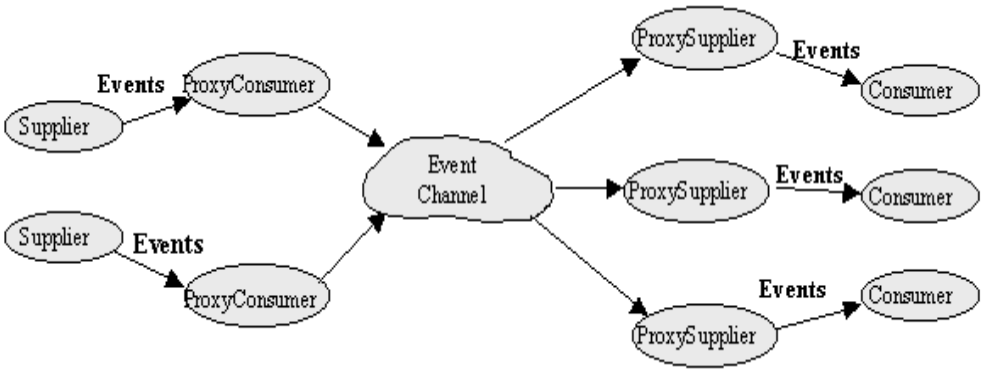
Esta ferramenta faz com que seja fácil expandir dinamicamente um agente SNMP em tempo de execução, já que os MIBs podem ser carregados e descarregados a qualquer altura. A implementação dos MIBs é totalmente independente da implementação do agente, e por isso também se torna fácil alterar a sua implementação durante a execução sem ter que recompilar o MIB.

## 7.4. Aplicações IDL e Object Request Broker

### 7.4.1. cosEvent

O serviço de eventos permite que os programadores subscrevam canais de informação. Os fornecedores de eventos podem gerar eventos sem sequer saberem a identidade dos consumidores bem como os consumidores podem receber eventos sem saber a identidade dos fornecedores.

Esta aplicação, CORBA Event Service, disponibiliza um modelo flexível de comunicações entre objectos de forma assíncrona.




• Figura 14 – Visão do funcionamento da aplicação CosEvent

- **Ø *Suppliers* e *Consumers*:** Os Consumidores (*Consumers*) são o salvos dos eventos gerados pelos Fornecedores (*Suppliers*). Ambos podem desempenhar papéis passivos ou activos. Há dois tipos de fornecedores e consumidores: *push* e *pull*. Um objecto *PushSupplier* pode fazer *push* a um evento para um *PushConsumer* passivo. De outra forma, um *PullSupplier* pode ficar passivamente à espera que um *PullConsumer* faça *pull* de um evento.

- ***EventChannel*:** Esta entidade desempenha um papel de mediador entre os consumidores e os fornecedores. Ambos registam os seus interesses aqui. Esta entidade disponibiliza comunicações de n para m. Esta entidade, que pode ser apenas denominada de canal, consome eventos de um ou mais fornecedores e fornece-os a um ou mais consumidores, mesmo que os fornecedores e consumidores usem modelos de comunicação diferentes.
- ***ProxySuppliers* e Proxy*Consumers*:** Os *ProxySuppliers* agem como intermediários entre os consumidores e o *EventChannel*. É semelhante a um *Supplier* normal mas inclui métodos adicionais para ligar um consumidor a si. Da mesma forma, um *ProxyConsummer* age como um intermediário entre os fornecedores e o *EventChannel*. É semelhante a um *Consumer* normal mas inclui métodos adicionais para se ligar a um fornecedor.
- ***Supplier* e *Consumer* Administrations:** Agem como fábricas para criar *ProxySuppliers* e Proxy*Consumers*.

### 7.4.2. cosEventDomain

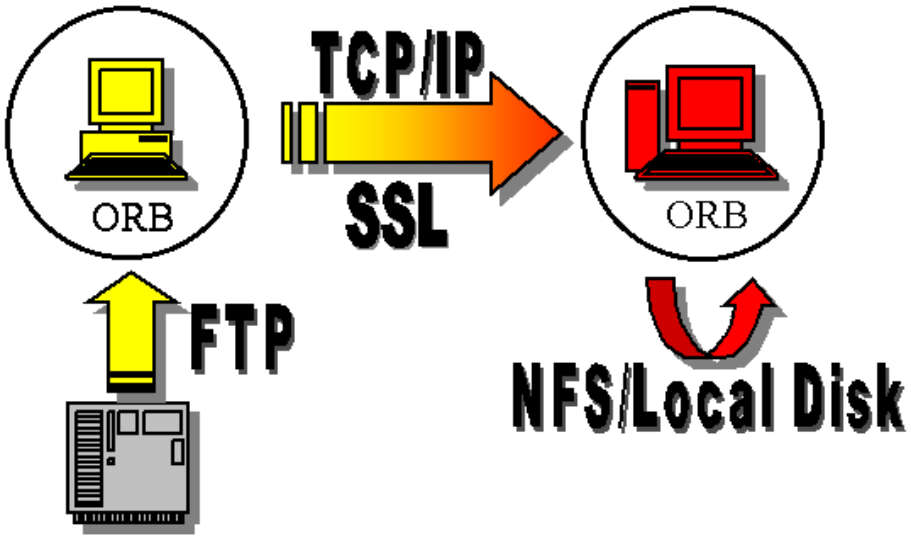
Este serviço permite que aos programadores gerirem uma série de canais de informações. É constituído por dois componentes:

- **EventDomainFactory:** uma fábrica para criar *EventDomains*.
- **EventDomain:** Disponibiliza uma ferramenta que torna fácil a criação de topo*log*ias de canais interligados.

### 7.4.3. cosFileTransfer

É uma aplicação para transferência de ficheiros. Esta aplicação depende das aplicações Orber, que disponibiliza as funcionalidades do CORBA no ambiente Erlang, e cosProperty.

Na figura seguinte vemos como esta aplicação funciona:



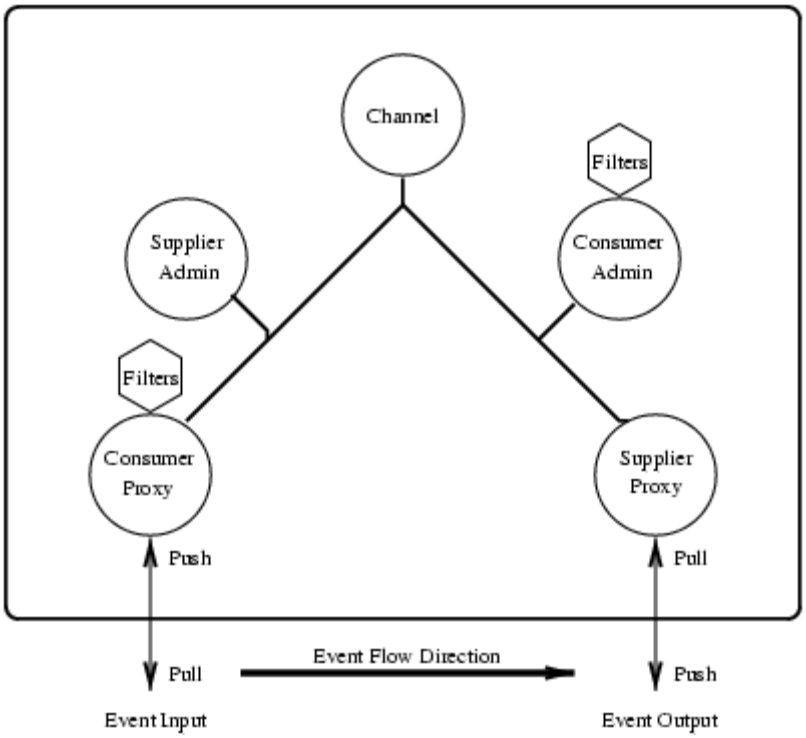

• Figura 15 – Visão do funcionamento da aplicação cosFileTransfer

- **Source ORB:** Representa o ORB do qual queremos transferir um ficheiro. Aloja um objecto referência para um VFS (*Virtual File System*), que neste exemplo é um servidor FTP.
- **Target ORB:** O objectivo pode ser transferir um novo ficheiro ou adicionar dados a um ficheiro já existente, localizado neste ORB. Neste exemplo é o disco local, ou o NFS.
- **Transport Protocol:** Inicialmente os dois ORBs comunicam usando pedidos CORBA para determinar se podem ou não comunicar. Se o serviço de transferência de ficheiros tiver um ou mais protocolos de transporte em comum, os dados vão ser transmitidos usando esse protocolo. Actualmente esta aplicação suporta TCP/IP e SSL.

### 7.4.4. cosNotification

É uma aplicação de notificação que depende das aplicações Orber v3.1.7 ou superior, cosTime v1.0.1 ou superior e de ficheiros IDL compilados com a aplicação IC v4.0.4 ou superior.

Esta aplicação é constituída por sete componentes, como descrito abaixo:




• Figura 16 – Visão do funcionamento da aplicação cosNotification

- **Event Channel:** Age como uma fábrica para objectos do tipo *Administrator*. Permite aos clientes definirem as propriedades administrativas.
- ***Supplier Administrators*:** Agem como uma fábrica para os Proxy*Consumers*
- ***Consumer Administrators*:** Agem da mesma forma que os *Supplier Administrators* mas em relação aos Proxy *Suppliers*.
- ***Consumer* Proxy:** Está ligado a uma aplicação cliente. Pode ser iniciado como um objecto *Pull* ou *Push*.
- ***Supplier* Proxy:** Age de forma semelhantes aos *Consumer* Proxy.
- **Filters:** usados para filtrar eventos.
- **Mapping Filters:** Usados para sobrepor as definições de QoS. Só podem ser associados a *Consumers Administrators* e a Proxy *Suppliers*.

### 7.4.5. cosProperty

É uma implementação em Erlang do serviço Corba Property.

### 7.4.6. cosTime

Esta aplicação usa a função `calendar:now_to_universal_time(Now)` para criar um UTC. O sistema operativo tem de fornecer um resultado correcto quando se chama a função `erlang:now()`. Esta aplicação é dependente do Orber. Usa como base a data 15 de Outubro de 1582 às 00:00h.

### 7.4.7. cosTransactions

Basicamente, esta aplicação implementa um protocolo de duas fases e permite que objectos a serem executados em plataformas diferentes, participem numa transacção.

### 7.4.8. IC

É um compilador IDL implementado em Erlang. Gera *stubs* e *skeletons*. São suportados diversos *back-ends*, agrupados em três categorias.

- O primeiro consiste num "*back-end*" CORBA.
  - IDL para Erlang CORBA – Serve para comunicações e implementações CORBA e o código gerado usa o protocolo CORBA específico para estabelecer a comunicação entre clientes e servidores.

- O segundo consiste num *back-end* Erlang.
  - IDL para Erlang – Providencia uma interface cliente em Erlang muito simples. Só pode ser usado dentro de nó Erlang, e como tal a comunicação entre cliente e servidor é feita apenas através da invocação de funções.
- O terceiro grupo consiste em *back-ends* para C, Erlang e Java. A comunicação entre os clientes e os servidores é feita através do protocolo de distribuição do Erlang. Todos os *back-ends* deste terceiro grupo geram código compatível com o protocolo de comportamento gen_server do Erlang.

- IDL para Erlang gen_server – são gerados *stubs* e *skeletons*. Os tipos de dados são mapeados de acordo com o mapeamento IDL para Erlang.
- IDL para cliente C – São gerados *stubs*.
- IDL para servidor C – São gerados *skeletons*.
- IDL to Java – São gerados *stubs* e *skeletons*.

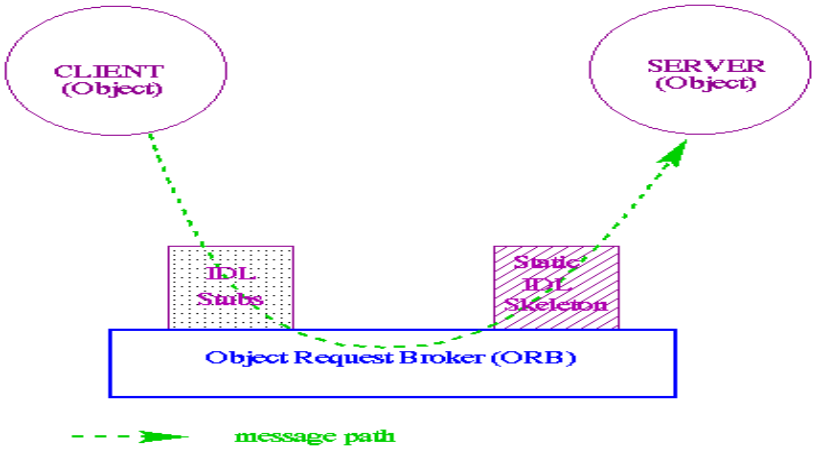
### 7.4.9. Orber

Pode ser descrito como um intermediário que cria relações entre clientes e servidores, definidas por interfaces. Isto permite uma maior transparência para os utilizadores, uma vez que estes não precisam de saber onde se encontra o objecto requisitado. Assim podem trabalhar com várias plataformas, desde que disponham de mapeamento IDL e interfaces.

O mapeamento IDL é o tradutor entre as diversas plataformas e linguagens. No entanto é o ORB que disponibiliza os objectos com a estrutura que possibilita a comunicação.

Um ORB intercepta e entrega-as a outros ORBs que por sua vez a entrega ao objecto destinatário.

Na figura vê-se como o ORB possibilita as comunicações:




• Figura 17 - Modelo ORB

## 7.5. Aplicações de Interface e Comunicações

### 7.5.1. Asn.1

Esta aplicação disponibiliza:

- Ø Um compilador ASN.1 para Erlang, que gera código e descodifica as funções que vão ser usadas pelos programas Erlang que enviam e recebem dados ASN.1.
- Ø Funções de *run-time* usadas pelo código gerado.
- Ø As regras de codificação suportadas são BER, DER e PER.

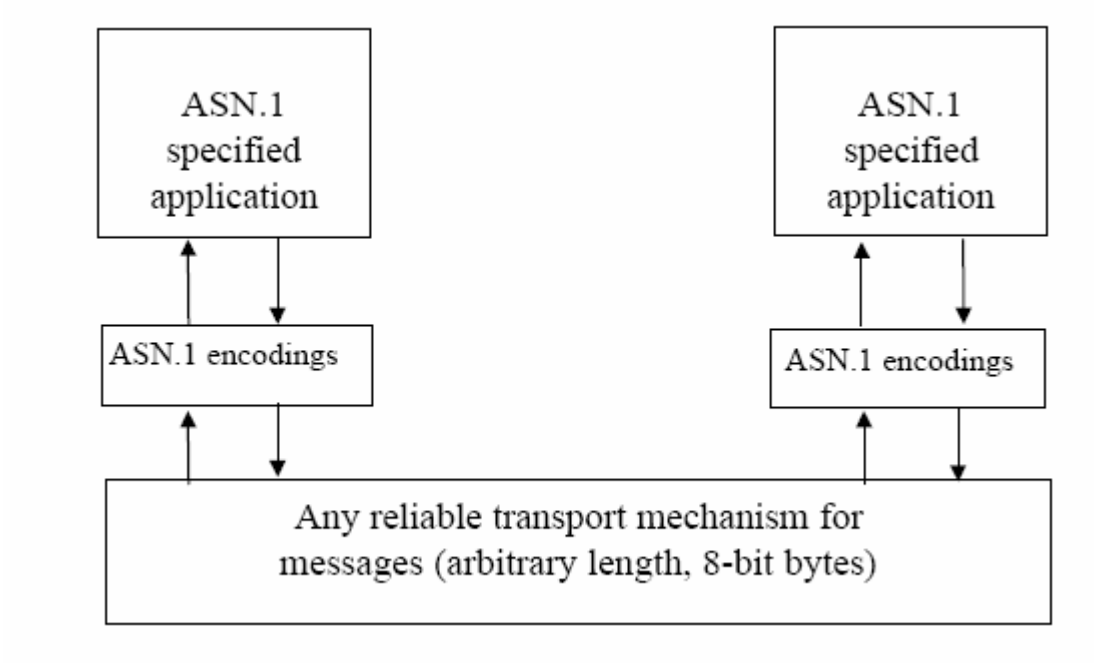
ASN.1 (*Abstract Syntax Notation One*) define uma sintaxe abstracta da informação. O objectivo desta notação é constituir uma linguagem independente da plataforma para definir tipos usando um conjunto de regras standard, que permitam transformar os dados de um determinado tipo num conjunto de bytes. Estes dados podem depois ser transmitidos através de protocolos como o TCP/IP e UDP. Desta forma, duas aplicações distintas, escritas em duas linguagens de programação distintas, a serem executadas em dois computadores distintos, com representações internas de dados distintas, podem trocar entre si instâncias de tipos de dados estruturados em vez de trocarem bytes.

Algumas das coisas que tornam esta notação importante são:

- Ser uma notação que é um standard internacional, independente da plataforma e da linguagem, para especificar estruturas de dados com um alto nível de abstracção.
- É suportada por um conjunto de regras que determinam os padrões de bits para representar os valores das estruturas de dados quando estes têm de ser transferidos através de uma rede.
- É suportada por ferramentas que estão disponíveis para a maioria das plataformas e para muitas linguagens de programação que mapeiam as estruturas de dados em notação ASN.1

A notação, e regras de codificação ASN.1 são usadas ao nível da camada de Aplicação e Apresentação do modelo OSI, respectivamente.

A notação ASN.1 permite especificar os tipos de dados e codifica-los de forma a poderem ser usados em diferentes plataformas, ou que programas em diferentes plataformas possam comunicar entre si.




• Figura 18 - Representação ASN.1

As regras de codificação ASN.1 são conjuntos de regras para transformar dados especificados na notação ASN.1 para um formato standard que possa ser descodificado em qualquer sistema que possua um descodificador baseado nas mesmas regras. Há vários conjuntos de regras de codificação:

- **BER** (Basic Encoding Rules) – este conjunto foi criado no início dos anos 80 e é usado numa vasta gama de aplicações, como o protocolo SNMP, Message Handlign Service (MHS) e TSAPI.
- **DER** (Distinguished Encoding Rules) – trata-se de uma especialização do BER, que é usado em aplicações com maiores necessidades de segurança, que envolvem operações de criptografia.
- **CER** (Canonical Encoding Rules) – é uma outra especialização do BER, semelhante ao DER, mas desenvolvido para ser usado na codificação de mensagens de grande volume de dados.
- **PER** (Packed Encoding Rules) – é um conjunto de regras mais recentes que os anteriores, e é reconhecido pelos seus algoritmos de

codificação que são mais rápidos e eficientes. Estas regras são usadas ao nível do controlo de tráfego aéreo e comunicações audiovisuais.

- **XER** (XML Encoding Rules) – permite que se codifiquem mensagens que foram definidas em ASN.1 através de XML.
- **E-XER** (Extended XER) – este conjunto de regras torna a notação ASN.1 num esquema tão poderoso como o XSD, mas com a simplicidade característica da notação ASN.1.

### 7.5.2. Crypto

O objectivo desta aplicação é disponibilizar algoritmos de hashing e encriptação para o SNMP. Em termos de algoritmos de hashing, são disponibilizados o MD5 e SHA. Para encriptação é usado o CBC-DES.

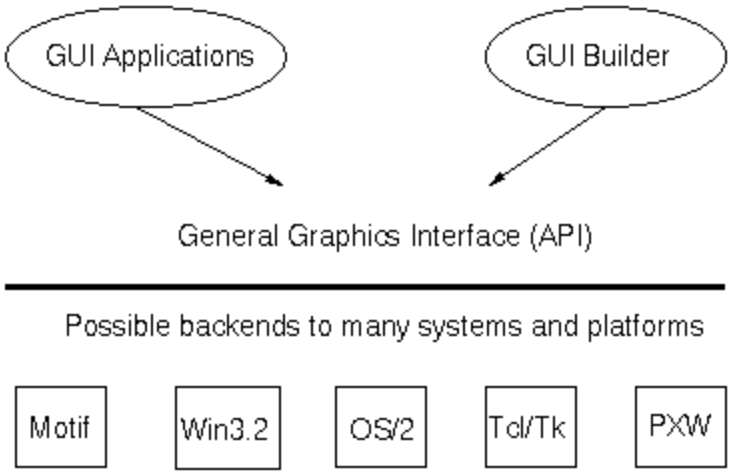
### 7.5.3. Erl_interface

Esta biblioteca contém funções que ajudam a integrar programas escritos em Erlang e C. Suporta as seguintes características:

- Manipulação de dados representados como sendo tipos de dados do Erlang.
- Conversão de dados entre os formatos do Erlang e o C.
- Codificação e descodificação de tipos de dados do Erlang para transmissão ou armazenamento.
- Comunicação entre nós Erlang e processos C.
- *Backup* e restauro do estado dos nós C através do Mnesia.

### 7.5.4. GS

É o sistema gráfico do Erlang. Foi desenhado de modo a ser fácil de aprender e de forma a ser um sistema gráfico portável entre diferentes plataformas. O Erlang foi implementado numa vasta gama de plataformas e o seu sistema gráfico funciona em todas elas.

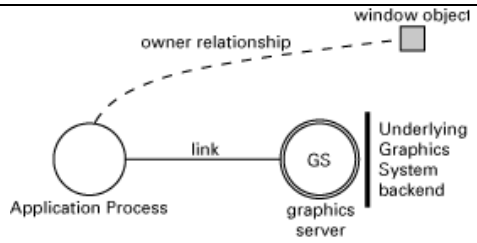



• Figura 19 – O Sistema Gráfico do Erlang

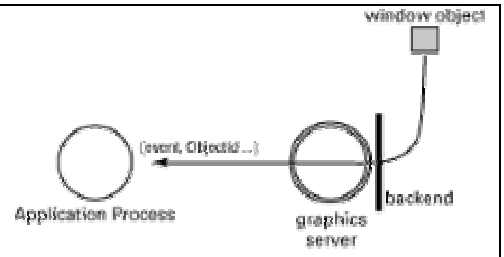
Os objectos desta aplicação são criados de uma forma hierárquica, em que cada objecto tem um objecto pai. Os tipos de objectos mais comuns são:

- window
- button
- label
- list box
- frame.

Sempre que um novo objecto é criado, é retornado um identificador de objecto. Através dele é possível configurar o objecto, alterando a sua aparecia e comportamento. Quando um processo Erlang cria um objecto gráfico, esse processo é o *owner* do objecto. Assim, o sistema gráfico (GS) tem de controlar os donos dos objectos para poder encaminhar correctamente os eventos respeitantes a cada objecto, bem como eliminar a janela que pertença a um processo que termine subitamente. Eventos são mensagens que são enviadas do objecto gráfico para o seu processo *owner*.




• Figura 20 – O Processo *Owner*




• Figura 21 – Eventos encaminhados para o processo Owner

### 7.5.5. Inets

É um contentor que disponibiliza um servidor http, uma versão eficiente do http 1.1, e um cliente FTP.

### 7.5.6. Jinterface

Este pacote disponibiliza um conjunto de ferramentas para as comunicações com processos Erlang. Também pode ser usado para comunicação processos Java, bem como com processos C, usando a biblioteca **Erl_Interface**.

O conjunto de classes deste pacote pode ser dividido em duas categorias: as classes que disponibilizam as comunicações e as classes que disponibilizam a representação em Java dos tipos de dados do Erlang.

Uma vez que este pacote disponibiliza mecanismos para estabelecer comunicações com o Erlang, os destinatários das mensagens podem ser processos Erlang ou instâncias de `com.ericsson.otp.erlang.OtpMbox`, sendo que ambos estão identificados com pids e possivelmente com nomes registrados. Estas classes suportam:

- Manipulação de dados representados como tipos do Erlang.
- Conversão de dados entre os formatos Java e Erlang.
- Codificação e descodificação de tipos de dados Erlang para transmissão e armazenamento.
- Comunicação entre nós Erlang e processos Java.

### 7.5.7. Megaco

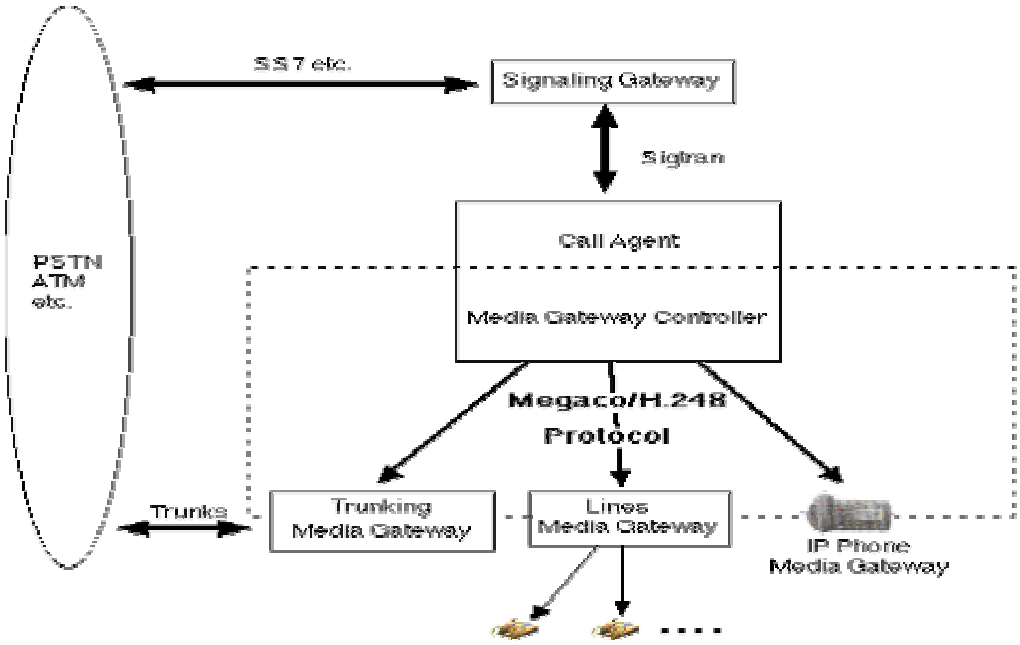
Megaco/H.248 é um protocolo para controlo de elementos num *gateway* multimédia decomposto, possibilitando uma separação entre controlo de chamadas e conversão de multimédia. Um MGC (*MEdia GAteway COntroller*) controla um ou mais MG (*Media Gateways*).

A semântica deste protocolo foi definida por duas organizações:

- IETF (Internet Engineering Task Force) – que trata o protocolo Megaco.
- ITU (International Telecommunication Union) – que trata o protocolo H.248.

Megaco é um protocolo do tipo *master*/*slave*, para controlo das funções de *gateway*. Tempos como exemplo o IP-PSTN *trunking gateway* e os *gateways* de linhas analógicas. A função principal do Megaco é permitir a decomposição do *gateway* em duas partes, o *master* que é um agente de chamada (MGC), e o *slave*, um interface de *gateway* (MC). O MG não tem qualquer conhecimento de controlo de chamadas.

SIP e H.323 são protocolos P2P para controlo de chamadas, ou mais genericamente, protocolos de sessões multimédia. Operam a diferentes níveis do Megaco numa rede decomposta e não têm conhecimento se o Megaco está ou não a ser usado em níveis inferiores.




• Figura 22 – Arquitectura de Rede

O Megaco possibilita:

- Ø Optimização de performance da rede.
- Ø Protecção de investimentos pelo isolamento das mudanças ao nível da camada do controlo de chamadas.
- Ø Liberdade para distribuir geograficamente as funções de chamada e de *gateway*.
- Ø Adaptação de equipamento antigo.

### 7.5.8. SSL

Uma interface Erlang para SSL (*Secure Sockets Layer*). Permite comunicações seguras sobre TCP/IP.

## 7.6. Aplicações de Ferramentas

### 7.6.1. Appmon

Esta aplicação é um nó gráfico que permite visualizar as aplicações que estão a ser executadas. Esta ferramenta exibe informação sobre todas as aplicações que estão a ser executadas em todos os nós conhecidos, sendo possível ver a árvore de processos de cada processo. No entanto se o código

desta aplicação não estiver presente num nó, este é ignorado e não há informações sobre os processos a correr nesse nó.

Para começar esta aplicação, usa-se a função `appmon:start()`, que lança a janela principal da aplicação, que mostra as aplicações em execução no nó local. Ao seleccionar uma dessas aplicações, será aberta a janela dessa aplicação. É possível ver informações sobre cada processo, enviar-lhe mensagens, fazer *trace* e até terminar o processo.

A janela é composta por menus que permitem:

- Sair do Appmon.
- Executar operações sobre o nó.
- Escolher o tipo de visualização pretendida.
- Seleccionar qual o nó a visualizar.
- Abrir a ajuda.

### 7.6.2. Debugger

É uma ferramenta gráfica usada para procurar erros e testar programas escritos em Erlang. Funciona como um *debugger* de outras linguagens, permite colocar *breakpoints*, mostra os valores das variáveis à medida que são alteradas, etc.

Para iniciar a aplicação, executa-se `debugger:start()`. Nesse momento é aberta uma janela com informação sobre todos os processos sujeitos a *debug*. No inicio não é costume haver qualquer processo, pois primeiro há que escolher quais os módulos a serem sujeitos a *debug*. Há a salientar que apenas os módulos compilados com a opção ***debug_*info** podem ser interpretados.

Quando um módulo é interpretado, pode ser visualizado numa janela desta aplicação. Assim pode-se ver o código fonte e estabelecer *breakpoints*. Depois já se pode iniciar o programa a que se quer fazer *debug*. Isto é feito normalmente através da *shell* do Erlang.

### 7.6.3. ET

Esta aplicação, ET (*Event Tracer*) usa o mecanismo de *trace* nativo do Erlang e disponibiliza ferramentas para uma visualização gráfica dos dados de *trace*.

Os dois maiores componentes desta aplicação são um visualizador de sequências gráficas chamado **et_viewer**, e o seu armazém de informações, chamado **et_*collector***. Um *collector* pode ser usado para vários *viewers* em simultâneo, em que cada um pode mostrar diferentes visões dos mesmos dados de *trace*.

#### 7.6.4. Observer

Aplicação que disponibiliza ferramentas para fazer *trace* a sistemas distribuídos.

#### 7.6.5. Parsetools

Contem ferramentas como o Yecc, que é um gerador de *parser* para Erlang semelhante ao **Yacc**. A partir de uma gramática, o gerador produz código Erlang para o *parser*.

#### 7.6.6. Pman

É um gestor de processos em modo gráfico, usado para inspeccionar os processos Erlang que estão a ser executados, tanto a nível local como em nós remotos. Também é possível fazer *trace* de eventos de processos individuais.

#### 7.6.7. Runtime_tools

Aplicação que disponibiliza ferramentas de baixo nível para *trace* e *debug*.

#### 7.6.8. Toolbar

A aplicação Toolbar oferece uma interface gráfica às várias ferramentas disponíveis do Erlang. Também pode permitir o acesso a ferramentas disponibilizadas pelo utilizador, a que se dá o nome de Contribuições GS.

Todas as ferramentas incluídas nesta aplicação têm de possuir um ficheiro de configuração que contenha informações sobre a ferramenta, tais como a sua função de início e a localização da informação de ajuda. O nome do ficheiro de configuração tem de possuir o sufixo **.tool**.

Para iniciar esta aplicação pode-se usar a função `toolbar:start()` na *shell* do Erlang, ou então iniciar o Erlang com o comando `erl -s toolbar`,

que faz com que a toolbar inicie ao mesmo tempo que a *shell*. Quando esta aplicação arranca, procura imediatamente por ficheiros com a extensão *.tool.

#### 7.6.9. Tools

Esta aplicação não é mais que um contentor que disponibiliza várias ferramentas stand-alone:

- **Cover** – Uma ferramenta de análise de coberturas para o Erlang.
- **Cprof** – Esta ferramenta mostra o número de vezes que uma função é executada.
- **erlang.el** – Modo Erlang para Emacs. Suporta funcionalidade de edição, destaques de acordo com a sintaxe, verificação do nome de módulos, suporte a comentários, etc.
- **Eprof e Fprof** – Ferramentas que mede a forma como o tempo é usado em programas Erlang.
- **Instrument** – funções uteis para obter e analisar a utilização de recursos num sistema *run-time* Erlang instrumentado.
- **Make** – Ferramenta similar ao *make* do UNIX.
- **Tags** – Ferramenta para gerar ficheiros TAGS do Emacs a partir de ficheiros de código fonte Erlang.
- **Xref** – Uma ferramenta de referência cruzada. Pode ser usada para verificar dependências entra funções, módulos, aplicações e *releases*.

#### 7.6.10. TV

Esta ferramenta examina graficamente as tabelas ETS e Mnesia. Assim que uma tabela é aberta nesta ferramenta, o seu conteúdo pode ser visto em vários níveis de detalhe. O conteúdo visualizado pode ser ordenado, usando um elemento como chave.

#### 7.6.11. webtool

É uma ferramenta usada para simplificar a implementação de ferramentas baseadas na web na plataforma Erlang/OTP.

# 8. Vantagens e Problemas do Elang/OTP

Esta plataforma pode ser executada em diversas plataformas de computadores e reúne aplicações que tanto podem ser escritas em Erlang como em outras linguagens.

Um dos grandes objectivos do OTP é providenciar um ambiente de desenvolvimento muito produtivo para o desenvolvimento de aplicações de telecomunicações. Este objectivo é conseguido à custa de:

- A linguagem Erlang.
- *Building blocks*.
- DBMS distribuídos de tempo real
- Bibliotecas para criação de aplicações que possam reportar erros, auto reiniciarem-se quando ocorrem erros e auto actualizarem-se com novas versões de software.
- Um agente SNMP e um servidor web que estão integrados com o DBMS.

Apenas uma pequena parte do suporte em tempo de execução do Erlang é dependente da plataforma do computador. A maior parte e todas as partes das aplicações que são escritas em Erlang são independentes do sistema de hardware e dos sistemas operativos.

Há que ter em atenção quando se usa software externo para se conseguir um bom suporte pois pode conter *bugs* que causam que o sistema se torne instável. Assim quando se usa software como um sistema operativo ou software que está directamente ligado ao hardware, deve ser software que esteja testado e isento de erros.

O OTP está desenhado para usar diferentes arquitecturas de processadores e *motherboards*.

As aplicações que usem OTP podem ser escritas em Erlang ou em qualquer outra linguagem.

No OTP, processos em diferentes nós comunicam entre si tão facilmente como processos num mesmo nó. No Erlang há a possibilidade de ligar processos, assim caso um processo termine, há a possibilidade dos outros serem avisados, para que se possa recriar o processo que terminou. Isto permite uma alta fiabilidade.

A Plataforma Erlang/OTP é especialmente recomendada para:

- Sistemas de Telecomunicações, para controlar switchs e conversão de protocolos.
- Aplicações de Telecomunicações.
- Servidores para aplicações Internet, como agentes, servidores IMAP ou HTTP.
- Aplicações de Base de Dados que requerem um comportamento *soft realtime*.

Isto acontece porque foi neste domínio que o Erlang foi desenvolvido inicialmente. As bibliotecas do OTP fornecem suporte para a maioria dos problemas em sistemas de telecomunicações.

O Erlang encontra maiores dificuldades em sistemas em que há factores constantes, cruciais para a performance, como processamento de imagens, de sinais, ordenação de grandes volumes de dados, drivers de dispositvos e terminações em protocolos de baixo nível.

O DBMS Mnesia tem grandes vantagens a nível de:

- Transacções – uma base de dados pode ser acedida por vários processos sem que seja necessário ao programador definir os controlos de acesso.
- Distribuição – as tabelas podem ser replicadas em diversos nós, por questões de eficiência e robustez.
- Dados não totalmente normalizados – ao contrário da maioria dos DBMS, os registos Mnesia podem conter dados de tamanho e estruturas arbitrários.
- Monitorização – os processos podem subscrever eventos que ocorrem aquando de operações sobre as tabelas.

A primeira intenção aquando da criação do Mnesia foi que fosse residente em memória, o que faz com que a informação seja guardada muito fragmentada quando trata grandes volumes de informação.

# 9. Desenvolvimentos Futuros

## 9.1. Integridade Conceptual

Desenvolver o Erlang de modo a reforçar a ideia que tudo é visto como processos. Tornar o sistema e o código Erlang mais regulares e mais fáceis de compreender.

## 9.2. Melhoramentos no Kernel

Um processo apesar de estarem isolados podem afectar a execução de outro processo do sistema ao alocar grandes quantidades de memória, ou enviar um grande número de mensagens. Um processo malicioso pode destruir um sistema inteiro apenas por criar grandes quantidades de átomos e com isso esgotar a capacidade da tabela de alocação de átomos. As implementações actuais do Erlang não estão desenhadas para proteger o

sistema deste tipo de ataques. São possíveis melhoramentos no Kernel que possam salvaguardar o sistema deste tipo de ataques.

## 9.3. Programação de Componentes

Se os processos comunicam uns com os outros, pode haver vantagens em escrever componentes em várias linguagens que comuniquem entre si.

# 10. Produtos Desenvolvidos em Erlang/OTP

## 10.1. Ericsson AXD301

O AXD301 é um *switch* ATM (*Asynchronous Transfer Mode*) produzido pela Ericsson. O sistema é constituído por um conjunto de módulos escaláveis, cada um capaz de disponibilizar uma capacidade de *switching* de 10Gb/s, que podem ser ligados até um máximo de 16 módulos.

Este *switch* foi desenhado para operações *"carrier-class"* sem interrupções. Dispõe de hardware em duplicado, e pode ser adicionado ou retirado sem haver necessidade de interromper o sistema. O software tem de ser capaz de lidar com falhas de hardware bem como de software, bem como ser possível alterar o software sem interromper o sistema.

Na tabela seguinte podemos ver a quantificação do sistema, a partir de um *snapshot* tirado em Dezembro de 2001.

| | |
|---|---|
| Numero de módulos Erlang | 2248 |

| | |
|---|---|
| Módulos "*Clean*" | 1472 |
| Módulos "*Dirty*" | 776 |
| Numero de linhas de código | 1136150 |
| Numero de funções Erlang | 57412 |
| Numero de funções "*Clean*" | 53322 |
| Numero de funções "*Dirty*" | 4090 |
| Percentagem de funções "*dirty*"/linhas de código | 0.359% |

• Tabela 1 – Quantificação do código Erlang no sistema AXD301

Uma função é classificada "*Dirty*" se enviar ou receber uma mensagem ou se chamar uma das seguintes BIF (*Built-In Functions*) do Erlang:

```
apply, cancel_timer, check_process_code, delete_module,
demonitor, disconnect_node, erase, group_leader, halt, link,
load_module, monitor_node, open_port, port_close,
port_command, port_control, process_flag, processes,
purge_module, put, register,
registered, resume_process, send_nosuspend, spawn, spawn_link,
spawn_opt, suspend_process, system_flag, trace, trace_info,
trace_pattern, unlink, unregister, yield.
```

### 10.1.1. Estrutura do Sistema

O código do AXD301 é estruturado usando as árvores de supervisão do Erlang. Os nós interiores da árvore são eles próprios nós supervisores, enquanto os nós exteriores são comportamentos OTP, ou processos específicos de aplicações especializadas.

A árvore de supervisão do sistema AXD tem 141 nós e usa 194 instâncias dos comportamentos OTP, de acordo com a distribuição seguinte:

| | |
|---|---|
| gen_server | 122 |
| gen_event | 36 |
| supervisor | 20 |
| gen_fsm | 10 |
| application | 6 |

• Tabela 2 – Distribuição de Comportamentos para o sistema AXD301

Os engenheiros do AXD301 definiram um supervisor *master* que podiam ser parametrizado de várias maneiras standard. Os supervisores deste

sistema eram empacotados como aplicações OTP convencionais, cujos comportamentos eram descritos nos ficheiros **.app**.

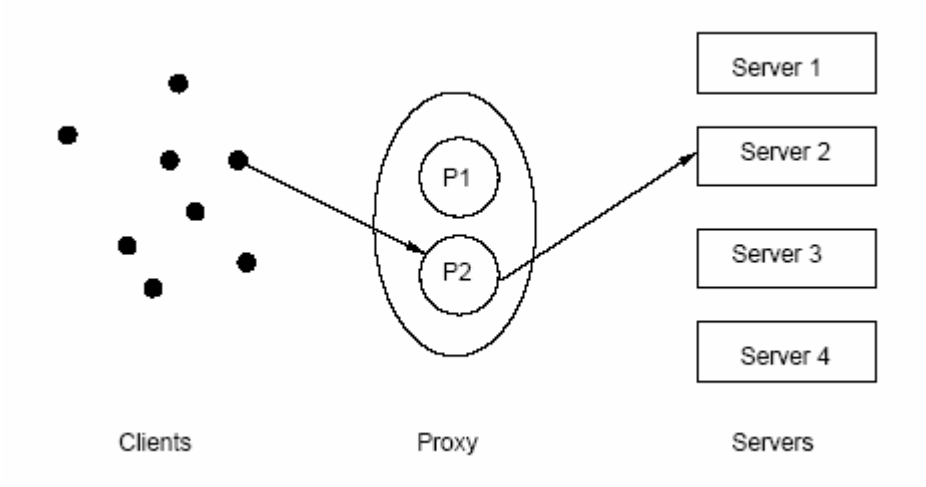
## 10.2. Bluetail Mail Robustifier

Este produto foi desenhado para aumentar a fiabilidade dos serviços de e-mail existentes. O BMR foi desenhado como sendo um proxy colocado entre os clientes que queriam usar um serviço de e-mail, e um número de servidores de e-mail.

Do ponto de vista dos clientes, todos os servidores de e-mail tinham o mesmo endereço IP, o endereço do proxy. Ao nível interno, este proxy tinha pelo menos duas máquinas, para ser tolerante a falhas. Os servidores de *back-end* eram eles próprios máquinas a executar servidores de e-mail standard. O BMR concentra-se em três protocolos de e-mail: SMTP, POP3 e IMAP4.

O BMR pode ser gerido remotamente e os *upgrades* ao sistema podem também ser executados remotamente. Foi escrito usando 108 módulos Erlang e tem 36.285 linhas de código. Foi desenvolvido de raiz e entregue ao primeiro cliente seis meses depois de iniciado o projecto. É usado pelo ISP Telenordia da Suécia desde 1999.

O BMR implementa o seu próprio sistema de gestão de *releases*, em extensão ao comportamento das *releases* no OTP.




• Figura 23 – Esquema de funcionamento do BMR

## 10.3. Alteon SSL accelerator

Este produto foi o primeiro a ser produzido depois da Bluetail AB ter sido adquirida pela *Alteon Web Systems*. Um acelerador SSL é uma aplicação de hardware contendo hardware específico para aumentar a velocidade operações de criptografia. O sistema de controlo para este acelerador é escrito em Erlang e foi desenvolvido em menos de um ano, tendo-se tornado num líder de mercado. A arquitectura de software usada foi derivada da arquitectura genérica usada no BMR.

### 10.3.1. Propriedades Quantitativas do código

| | |
|---|---|
| Numero de módulos Erlang | 253 |
| Módulos "*Clean*" | 122 |
| Módulos "*Dirty*" | 131 |
| Numero de linhas de código | 74440 |
| Numero de funções Erlang | 6876 |
| Numero de funções "*Clean*" | 6266 |
| Numero de funções "*Dirty*" | 610 |
| Percentagem de funções "*dirty*"/linhas de código | 0.82% |

• Tabela 3 – Quantificação do código Erlang no sistema Alteon SSL accelerator

São usados 103 comportamentos OTP, de acordo com a distribuição seguinte:

| | |
|---|---|
| gen_server | 56 |
| supervisor | 19 |
| application | 15 |
| gen_event | 9 |
| rpc_server | 2 |
| gen_fsm | 1 |
| supervisor_bridge | 1 |

• Tabela 4 – Distribuição de Comportamentos para o sistema Alteon SSL accelerator

# 11. Conclusão

A plataforma Erlang/OTP e a tecnologia associada é funcional, apesar de haver muita gente que clama que linguagens funcionais como Erlang não podem ser usadas para desenvolvimento de software a nível industrial. Como prova em contrário temos os exemplos do AXD301 e duma vasta gama de produtos da Nortel, entre os quais o *Alteon SSL Accelerator*, que são líderes de mercado.

O Erlang é uma linguagem de programação que utiliza processos leves e sem memória partilhada, que funciona na prática e pode ser usada para produção de software complexo a grande escala. É possível através desta plataforma construir sistemas distribuídos de alta fiabilidade e tolerantes a falhas.

O OTP permite aos programadores construir e executar aplicações de telecomunicações numa vasta gama de plataformas de hardware e software

standard. Permite que programadores que programem em C, C++, Java e outras linguagens integrem componentes OTP nas suas aplicações.

O OTP vem com uma colecção exaustiva de ferramentas e *bulding blocks*, dos quais se destacam a linguagem Erlang, SASL, um DBMS de tempo real, um agente SNMP e um servidor web.

Como ambiente de desenvolvimento, as maiores vantagens do OTP são *time to market*, compatibilidade com muitas e diversas plataformas de computadores e componentes, e uma grande fiabilidade.

Como um *target enviroment*, o OTP faz face a todos os requisitos de telecomunicações. Tem sistemas de controlo distribuídos de tempo real, é tolerável a falhas, e suporta *updates* de software enquanto está a ser executado. E não esquecer que pode ser portado entre diversas sistemas operativos.

# 12. Bibliografia


- http://www.erlang.se
- http://www.erlang.org
- Tese de Doutoramento de Joe Armstrong, de Dezembro de 2003
- Däcker, B.; Erlang – A new programming language. Ericsson Review 70 (1993:2)