Design Principles

version 4.8

OTP Team

1997-05-02

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

Design Principles

This chapter describes the software architecture used in an Erlang system. The first section describes what is meant by a system, how systems are composed of applications, and how application are built using *behaviours*.

The following four sections describe individual behaviours in detail. These behaviours include:

- supervision
- servers
- events
- finite state machines

The design principles are then summarized and numerous examples illustrate the tasks involved in developing Erlang applications. These include:

- how to structure an application
- how to create a supervision tree
- how to use the common behaviours
- how to install event handlers
- how to configure an application
- how to write an application specification
- how to test an application
- how to write a distributed application.

The final two sections describe special processes and error logging. Special processes contain information necessary to write specific applications which are not built from the standard system building blocks (behaviours), but are programmed in some other style. Such applications still have to follow certain basic protocols to in order to work with the rest of the system.

1.1 Overview of Design Principles

Systems, or complete products, are made from a number of Applications [page 5]. Applications provide the basic packaging mechanism for delivering systems. Applications are designed to be "weakly coupled" and it is often possible to make systems by combining existing applications with your own special purpose applications. New applications should be designed to be self sufficient, so they can be added to the existing base of applications and offered to future users of the system.

Examples of existing applications are `mnesia`, which has everything needed for programming database services, and the `gs` graphics system for building graphical user interfaces.

Applications are specified in terms of resources. Resources include modules, registered names, processes, and things like dependencies on other applications.

Applications must obey certain laws and must follow certain protocols so that they present a uniform interface to the Erlang system. For example, they must be written so that the code can be changed without stopping the system.

The easiest way to program a new application is to make use of the behaviours which are included in the system. A behaviour is a "pattern of design" which can be used to build an application. Applications which are programmed with the standard behaviours automatically follow the required protocols. Behaviours are explained in the next section of this chapter.

The most common way of programming an Erlang application is to start with a supervision tree [page 21]. A supervision tree is a hierarchical tree of processes used to program fault-tolerant systems. The higher nodes in the supervision tree are called *supervisors*. They monitor the lower nodes, which are called *workers*, and detect when failures occur in the lower nodes.

Worker nodes actually perform computations. They do the work, the supervisors only check the status of the work and restart them if things go wrong. This supervision principle makes it possible to design and program fault-tolerant software. Worker nodes should also be programmed using behaviours, but this depends on what the worker nodes have to do.

If an application is written without the help of the behaviour modules, then the programmer must ensure that the application follows the required protocols. The following two modules are provided to help program applications which do not make use of the standard behaviors:

- `sys`. This module provides a set of library functions that follow the standard system protocols. Functions in `sys` should be used to interface your process to the rest of the system.
- `supervisor_bridge`. This module makes it possible to use an existing set of processes within a supervision tree.

Both `sys` and `supervisor_bridge` are intended for somewhat specialized usage and require detailed knowledge of the Erlang system.

When writing code that is called by the standard behaviours, the programmer can call functions in the standard libraries. The libraries provide a rich and growing set of modules which contain commonly used library functions, such as `lists`, `strings`, `ordsets`, `dict`, and `file`.

Behaviours

Behaviours are formalizations of “design patterns” which can be used to program certain common problems.

Concurrent systems can be programmed by combining ideas and code from a small number of design patterns. Each design pattern, which we call a behaviour, solves a particular problem.

The standard behaviours which are included with the system are:

- `application`. This behaviour defines how applications are implemented and terminated.
- `sup_bridge`. This behaviour [page 26] is used to connect a process, or subsystem, to a supervisor tree although it has not been designed with the supervision principle in mind.
- `supervisor`. This behaviour [page 21] is a worker/supervisor model for structuring fault tolerant computations, and for programming supervision trees.
- `gen_server`. This behaviour [page 29] is used for programming client-server processes.
- `gen_event`. This behaviour [page 37] is used for programming event handling mechanisms, such as event handlers and loggers, error loggers, and plug-and-play handlers.
- `gen_fsm`. This behaviour [page 49] is used for programming finite state machines.

Behaviours are implemented as *callback modules*. A callback module must export a specific set of functions, which are then called by the system as the behaviour process executes.

All modules which make use of behaviours should start as follows:

```
-module(xx).
-behaviour(yy).
...
```

This means that the module `xx` has the behaviour `yy`. In the following declaration, the module `disk_alloc` has behaviour `gen_server`

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

In a following section, titled Servers [page 29], the following type of statements are used (see also the Reference Manual, `stdlib`, the module `gen_server`):

```
Module:init(Args) -> {ok, State} | {ok, State, Timeout}|
                        ignore | {stop, StopReason}
   ...
Module:handle_call(Request, From, State) -> CallReply
   ...
```

Here `Module` is the name of the callback module. In the previous example, this is the module `disk_alloc`, and `disk_alloc` must export `init/1` and `handle_call/3` for example.

Read the section Servers [page 29] in this chapter, which explains the philosophy of the client-server behaviour. This section should be read in conjunction with the Reference Manual, `stdlib`, module `gen_server`, which describes the callback API in greater detail.

It is sometimes possible to write a program which does not make use of the application and behaviour mechanisms. Such programs may be more efficient, but the increased efficiency will be at the expense

of generality. The ability to manage all applications in the system in a consistent manner is very important.

Programmers will find it easy to read and understand the code produced by others who are familiar with the application architecture and standard behaviours. Ad hoc programming structures, while possibly more efficient, are always more difficult to understand.

1.2 Applications

Applications are used for “packaging” system components. An application consists of a set of resources, such as modules, registered names, and processes.

By structuring the system into a well-defined set of applications, the system designer is forced to think about the system in terms of its sub-components and to decide which functionality each application should have.

Programming an Application

Several operations are possible with an application. In particular, we can load, unload, start and stop an application.

When an application is loaded, the system checks that all the resources the application will need are present. If loading is successful, the application can be started at a later time.

When an application is started, an `application_master` process is created. Application master is the group leader of all the processes in the application.

The application master is aware of every process in the application. Thus, if the application was stopped the application master would terminate all processes, for which this application is responsible, in a controlled fashion.

When an application is unloaded, all code relating to the application is removed.

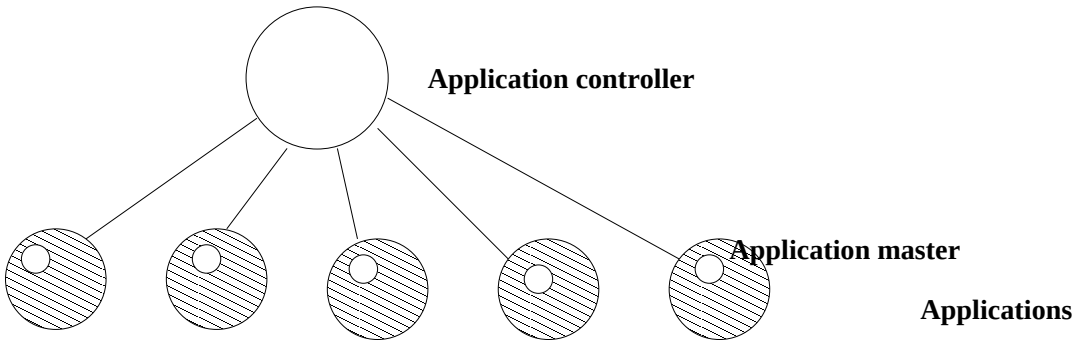


Figure 1.1: Application Controller and Applications

All operations on applications are coordinated by a single process with the name `application_controller`. The shaded circles in the above diagram represent applications. Each individual application is controlled by an application master.

The Application Resource File

The resources required by each application are defined in an application resource file. These files have the extension `.app`, which specifies the resources required by the application and how the application should be started.

A resource file (`ApplName.app`) has the following syntax:

```
{application, ApplName,
  [{description,  String},
   {vsn,          String},
   {id,           String},
   {modules,      [{Mod1,Vsn1}, Mod2, {Mod3,Vsn3} .., {ModN,VsnN}]},
   {maxP,         Int | infinity},
   {maxT,         Seconds | infinity},
   {registered,   [Name1, Name2, ...]},
   {applications, [Appl1, Appl2, .., ApplN]},
   {included_applications, [Appl1, Appl2, .., ApplN]},
   {env,          [{Par1, Val1}, {Par2, Val2} .., {ParN, ValN}]},
   {mod,          {Mod, StartArgs}},
   {start_phases, [{Phase, PhaseArgs}]}]}.
```

The keys have the following meanings:

- `description`
  Textual description of the application.
- `vsn`
  Version of the application. It should not contain a directory separator.
- `id`
  Product identification of the application.
- `modules`
  List of all the modules and their versions introduced by this application. A module can be listed without a version, with only the module name stated. A module can only be defined in one application.
- `maxP`
  Maximum allowed number of simultaneous processes which this application can manage (or the atom `infinity`). The key `maxP` is optional and defaults to `infinity`.
- `maxT`
  Maximum time that the application can run (or the atom `infinity`). The key `maxT` is optional and defaults to `infinity`.
- `registered`
  Lists all the names of registered processes used in this application.
- `applications`
  List of applications which must be started before this application can be started. Most applications have dependencies to the `kernel` and `stdlib` applications.
- `included_applications`
  List of applications which are included by this application. An included application is loaded, but not started, by the `application_controller`. Processes implemented in an included application should be placed underneath a supervisor in the including application. This key is optional and defaults to an empty list.

- `env`
  List of the environment variables in the application. Each parameter `ParX` is an atom, and the associated `ValX` can be any term. The `env` key is optional and defaults to an empty list.
- `mod`
  Application call back module of the application behavior. The *application master* starts the application by evaluating the function `Mod:start(Type, StartArgs)`, refer to the kernel reference manual. When the application has stopped, by command or because it terminates, the *application master* calls `Mod:stop(State)` to let the application clean up. If no `State` was returned from `Mod:start/2`, then `Mod:stop([])` is called. The `mod` key should be omitted for applications which are code libraries, such as the `stdlib` application. These applications have no dynamic behavior of their own and should not have a start function.
- `start_phases`
  List of start phases and the attached start arguments for the application. The *application master* starts the application by evaluating the function `Mod:start_phase(Phase, Type,PhaseArgs)`, for each defined start phase. (See also the Kernel reference manual). Each parameter `Phase` is an atom, and the associated `PhaseArgs` is a list of any terms. The key `start_phases` is optional, but the behavior of the system is dependent on the key being defined or not (see kernel reference manual and the `Starting Applications` chapter below).

The Application Directory

Each application should be placed in a separate directory. This directory should be divided into several sub-directories. The following sub-directories should exist in the application directory `.../ApplName`:

- `src`
- `ebin`
- `priv`
- `include`

This structure is necessary because tools such as `systools` rely on this structure to be able to generate boot scripts and release packages. The release handling procedure on target machines also depends on this structure to be able to upgrade to new releases.

Note:
In the target environment, the application directory name contains the application version.

The following description of the sub-directories listed also indicates if the sub-directory is mandatory or optional.

- `src`.
  This directory contains the source code. This directory is mandatory in the development environment, but optional in the target environment. If source code is written in several different languages, a sub-directory with the name `e_src` can be created below the `src` directory to store the Erlang source code.
- `ebin`.
  This directory contains the Erlang object code, for example `jam` files. The application resource file should also be placed here. This directory is mandatory.
- `priv`.
  This directory is used for application specific files. For example, `C` executables should be placed here, or in a `bin` directory below the `priv` sub-directory. The application designer can determine the directory structure below `priv`. The function `code:priv_dir/1` should be used to access the `priv` sub-directory. This directory is optional. It can be omitted if the application does not include any application specific files.
- `include`.
  Included files must be placed in this directory. The application exports these files for inclusion in other application modules. The `include_lib("ApplName/include/File.hrl")` module attribute should be used to include a file from another application. This directory is optional.

Configuring an Application

Application specific configuration parameters should be specified in the `.env` key in the resource file [page 6]. The application can then call `application:get_env(ApplName, Parameter)` to retrieve the values for the configuration parameters. You can also override configuration parameters by using the *system configuration file*. This file is specified with the command line parameter `-config FileName`. The file `FileName.config` contains a list of configuration parameters for the applications. For example:

```
[{sasl, [{sasl_error_logger, tty},
         {errlog_type, error},
         {disk_space_check_interval, 10}]},
 ...
 {ApplNameN, [{Par1, Val1},
              {Par2, Val2}]}].
```

The parameters over ride can also be executed from the command line:

```
erl -ApplName Par1 Val1 Par2 Val2 ...
```

Note:
Each term should be an Erlang term. However, in the Unix shell, the term must be enclosed in single quotation marks. For example: `'{file, "a.log"}'`.

The Application Master

Each individual application is controlled by an *application master*.

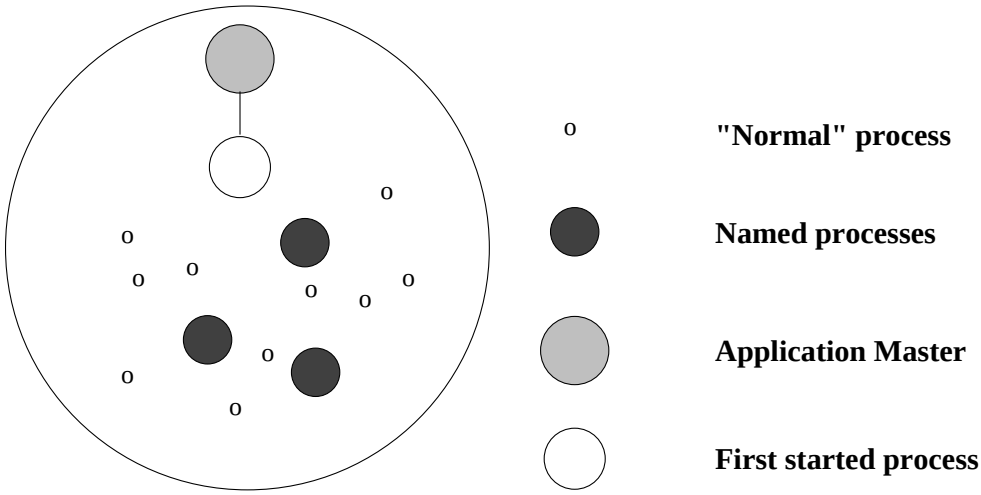


Figure 1.2: Example of an Application

The *application master* is responsible for all processes running in an application. The *application master* can kill all processes which belong to the application.

The *application master* assumes that the process started by the start function is responsible for the internal details of the application. This process is assigned the following special role:

- If this process terminates, then the *application master* assumes that the application as a whole has terminated.
- If the process started first terminates with a normal exit, then the application is assumed to have terminated correctly.
- If the process started first terminates with an abnormal exit, then the application is assumed to have terminated with an error.

An application can be started in one of three modes: permanent, transient, or temporary. Default value is temporary. The mode specifies what happens if the application dies.

- If a permanent application dies, all other applications are also terminated.
- If a transient application dies normally, this is reported and no other applications are terminated. If a transient application dies abnormally, all other applications are also terminated.
- If a temporary application dies this is reported and no other applications are terminated. In this way, an application can run in test mode, without disturbing the other applications.

Included Applications

An application can *include* other applications. Processes implemented in an included application should be placed underneath a supervisor in the including application. This way, you include applications in order to group them together.

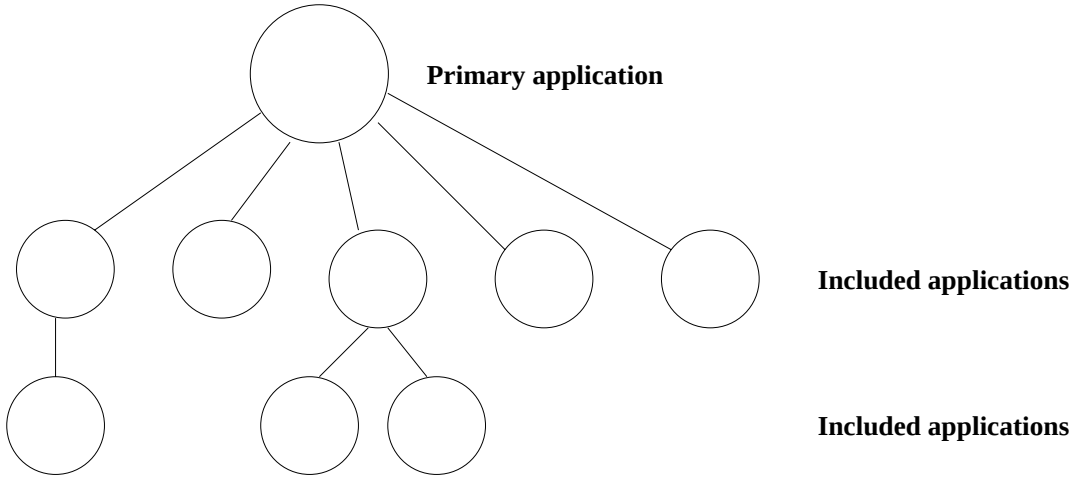


Figure 1.3: Primary Application and Included Applications

An application which is not included by any other application is called a primary application. An application can only be included by one other application. If you want several applications to include an application, it has to be designed as a library application without a start function (the `mod` key in the .app file).

When an application is loaded, the application controller ensures that all included applications are also loaded. When an application is started , the application controller will only start the primary application by default (see chapter, `Starting Applications`).

> Note:
> The information in the following paragraph only applies to the Erlang System/Open Telecom Platform (OTP) environment.

When building a release, the `included_applications` key, which is specified in the application resource file, can be overridden with the `.rel` file. Therefore, all applications have the implicit environment (`env`) variable `included_applications` in order to read the current configuration. This is

useful when an application has multiple start parts (included applications) depending on customer requirements, and where there is no need install or load excluded parts in the system. Refer to the section Release Structure in the System Architecture Support Libraries Manual for more details.

Distributed Applications

In a distributed system with several cooperating Erlang nodes, there is a need to control certain applications in a distributed manner. Some applications may be specified to run on one of several nodes. The application controllers on the different nodes can be arranged to monitor each other. If one node goes down, the application controller on another node notices this and restart the application on its node. These applications are called *distributed* applications, as opposed to *local* applications, which are always started on the local node. An example of a standard local application is `kernel`; there is always one local instance of `kernel` running on an Erlang node.

With this definition, a local application may be distributed in the sense that it uses services on other nodes, or cooperates with applications on other nodes. For example, the application controller sees the Mnesia DBMS started as a local application. In this description, only the *control* of applications is discussed.

Because a distributed application may move between nodes, some addressing mechanism is required to ensure that it can be addressed by other applications, regardless on which node it currently executes. This issue is not addressed here, but the standard Erlang modules `global` or `pg` can be used for this purpose.

Specifying Distributed Applications

The configuration parameter `distributed` for the application `kernel` defines which applications are distributed, and on which nodes they may run. This parameter is of the form `{ApplName,[NodeDesc]}` or `{ApplName, Time, [NodeDesc]}`, where `NodeDesc = Node |{Node, ..., Node}`. This data structure specifies a list of nodes where the application `ApplName` may execute, and the order in which these nodes should be used to start the application. If the nodes are specified in a tuple, the order is undefined. If a node crashes and `Time` has been specified, then the application controller will wait for `Time` milliseconds before attempting to restart the application on another node. If `Time` is not specified, it defaults to 0, and if a node goes down the application is restarted immediately on another node.

For example, suppose that the application `myapp` is defined to be distributed as `{myapp, 5000, [cp1@cave, {cp2@cave, cp3@cave}]}`. Suppose further that all nodes are up and running when `myapp` is started. It is then started at `cp1`, as shown in the figure below.

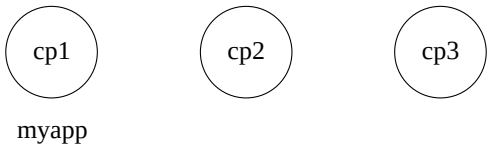


Figure 1.4: Application myapp - Situation 1

If `cp1` goes down, the system checks which one of the other nodes, `cp2` or `cp3`, has the least number of running applications, but waits for 5 seconds for `cp1` to restart. If `cp1` does not restart and `cp2` runs fewer applications than `cp3`, then `myapp` is restarted on `cp2`.

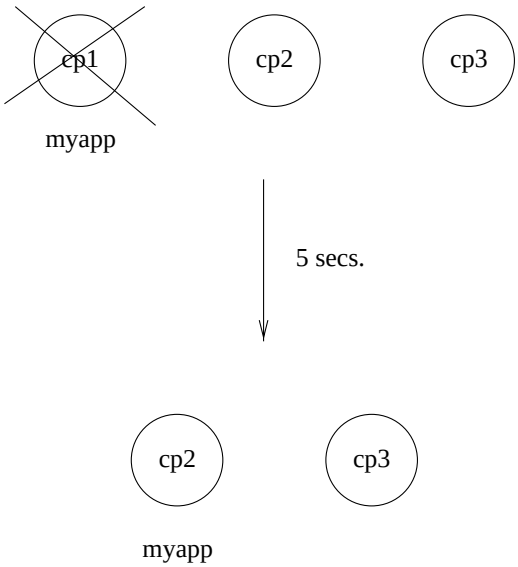


Figure 1.5: Application myapp - Situation 2

Suppose now that cp2 goes down as well and does not restart within 5 seconds. myapp is now restarted on cp3.

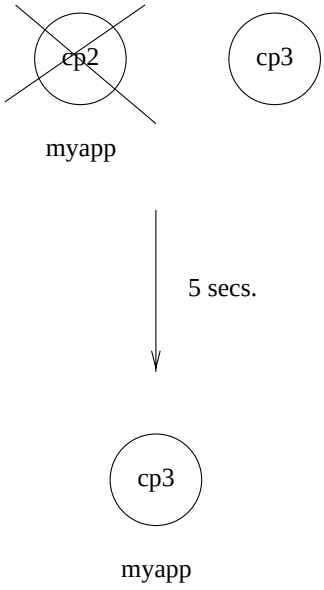


Figure 1.6: Application myapp - Situation 3

If cp2 now restarts, it will not restart myapp, because the order between nodes cp2 and cp3 is not defined.

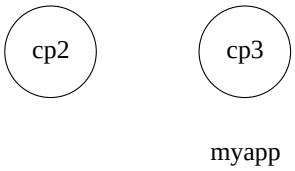


Figure 1.7: Application myapp - Situation 4

However, if `cp1` restarts as well, the function `application:takeover/2` moves `myapp` to `cp1`, because `cp1` has a higher priority than `cp3` for this application. In this case, `Mod:start({takeover, cp3@cave}, StartArgs)` is executed at `cp1`.

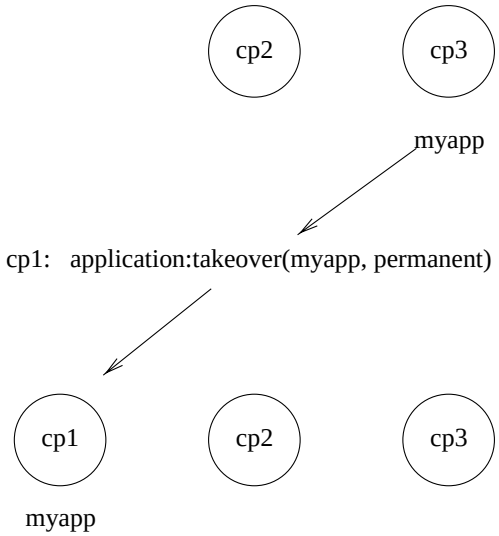


Figure 1.8: Application myapp - Situation 5

For a distributed application, the `Nodes` list distribution specification must be the same at each involved node. The application controllers check this when they contact each other at start-up. If a mismatch is found, the application controller, and thus the entire Erlang node, terminates with reason `{distribution_mismatch, ApplName, Node}`. The reason for this behavior is that the application may not start at all, it may start at several nodes, or the application controller may be left hanging if there is a mismatch in the specification.

The `distributed` parameter can be changed at release upgrade. The applications are however not automatically moved to the nodes which have the (new) highest priority; this has to be made manually by the operator after the release upgrade.

Starting Distributed Applications

Each node which is included in the distribution specification for a distributed application must make an `application:start` call for the distributed application. The start-up sequence between local and distributed applications is also synchronized.

To illustrate this point, suppose that a local application named `localapp` uses the distributed application `myapp`, described in the previous example, with the distribution specification `{myapp, 5000, [cp1@cave, {cp2@cave, cp3@cave}]}`. It is assured that `localapp` will not be started on `cp1`, `cp2` or `cp3` until `cp1` starts `myapp`. In this way, the start of distributed applications is a synchronization point for all nodes that may run the application.

When a local application is dependent on a distributed application but is configured to run on a different node than the distributed application a synchronization problem occurs, this is solved by also starting the distributed application on the node where the local application will run. This ensures that the applications are started in the correct order on the right nodes.

Please note when the above configuration applies the local application must be loaded before the distributed application is started. Normally this is not a problem because all applications are loaded before any applications are started.

When distributed applications are specified, it is advised that the `sync_nodes` functionality in the `kernel` application is used. If not used, and several nodes are started at the same time, the node that comes up first will start all applications, and the other nodes will take over the applications when they come up.

Starting Applications

A primary application can be started in one or two steps. The first step is mandatory and the purpose of it is to start the main supervisor of the application and possible permanent children. The second step is optional and its purpose is to synchronize processes within an application.

Start Supervisors

The start parameters for the first step are defined in the `mod` key in the resource file for the primary application. The *application master* will evaluate the function `Mod:start(Type, StartArgs)`.

`Mod` and `StartArgs` parameters are defined in the `mod` key. The parameter `Type` states what type of start is running, ie. normal, takeover or fail-over.

Takeover signifies that the application is distributed and is to be moved from another node to this node, due to operator invention or due to this (newly started) node is defined as superior to the other node in the configuration parameter `distributed`.

Fail-over signifies that this node should start the application due to a crash of the node where the application was previously running.

All other starts will result in a normal start type.

Note: Fail-over is only valid if the `start_phases` key is defined, otherwise this start type is denoted as normal. If older application versions without start phases are being used, it is possible to set the `start_phases` key to an empty list.

Synchronize Processes

The second step is executed only if the `start_phases` key is defined in the resource file of the primary application.

If a primary application has a defined key, all its included applications must also have the key defined. There is, however, a possibility to include applications without start phases by wrapping them in an another application (how to do this is explained later).
In this step the *application master* evaluates the function `Mod:start_phase(Phase, Type, PhaseArgs)` for each phase in the specified order. `Mod` is defined in the `mod` key, `Phase` and `PhaseArgs` in the `start_phases` key, and `Type` is the same as in the first step. An example showing (a part of) an application's resource file and the evaluated functions:

myApp.app

```
:
{mod, {myApp, StartArgs},
{included_applications, []},
{start_phases, [{init, InitArgs}, {go, GoArgs}]},
:
```

evaluated functions

```
myApp:start(Type, StartArgs)                % start the main supervisor
                                            % and permanent children
myApp:start_phase(init, Type, InitArgs)     % start the application
                                            % in the init phase
myApp:start_phase(go, Type, GoArgs)         % start the application
                                            % in the go phase
```

A primary application is responsible for starting included applications. The primary application can read the included applications start phases by calling `application:get_key(Application, start_phases)`. There is, however, a possibility to automate the start of included applications, but first an example where the synchronization is taken care by the primary application:

primApp.app

```
:
{mod, {primApp, PrimAppStartArgs},
{included_applications, [inclOne, inclTwo]},
{start_phases, [{init, InitArgs}, {go, GoArgs}]},
:
```

inclOne.app

```
:
{mod, {inclOne, NotUsedArgs},
{start_phases, [{go, GoArgs1}]},
:
```

inclTwo.app

```
:
{mod, {inclTwo, NotUsedArgs},
{start_phases, [{init, InitArgs2}, {go, GoArgs2}]},
:
```

```
evaluated functions
primApp:start(Type, PrimAppStartArgs)       % start the main supervisor
                                            % and permanent children

primApp:start_phase(init, Type, InitArgs)   % start the PrimApp and InclTwo
                                            % applications in init phase
primApp:start_phase(go, Type, GoArgs)       % start all applications
                                            % in go phase
```

To automate the synchronization of the included applications a predefined module application_starter is to be used as the Module in the mod key. application_starter takes the Mod and StartArgs as parameters: {mod, {application_starter, [{Mod, StartArgs}]}.

It is possible to have unique start phases in different applications. The order of the phases is defined in the primary applications start_phases key for the whole tree. This aserts that all included applications must have their start phases defined in the including applications' resource files.

The application_starter will execute only one start phases at a time, from left to right, searching and executing (where indicated) start phases in the included applications.

The above example is continued below using an automatic start for included applications:

```
primApp.app
:
{mod, {application_starter, [primApp, PrimAppStartArgs]},
{included_applications, [inclOne, inclTwo]},
{start_phases, [{init, InitArgsPrim}, {go, GoArgsPrim}]},
:

inclOne.app
:
{mod, {inclOne, NotUsedArgs},
{start_phases, [{go, GoArgs1}]},
:

inclTwo.app
:
{mod, {inclTwo, NotUsedArgs},
{start_phases, [{init, InitArgs2}, {go, GoArgs2}]},
:


evaluated functions
primApp:start(Type, PrimAppStartArgs)         % start the main supervisor
                                              % and permanent children
primApp:start_phase(init, Type, InitArgsPrim) % start the primApp application
                                              % in the init phase
inclTwo:start_phase(init, Type, InitArgs2)    % start the inclTwo application
                                              % in the init phase
primApp:start_phase(go, Type, GoArgsPrim)     % start the primApp application
                                              % in the go phase
inclOne:start_phase(go, Type, GoArgs1)        % start the inclOne application
```

```
                                                % in the go phase
inclTwo:start_phase(go, Type, GoArgs2)          % start the inclTwo application
                                                % in the go phase
```

For the reason mentioned above, the subsequently included applications on the same branch must have their start phases defined in *all* the including applications resource files (`start_phases` key). The `application_starter` will then run the start phases in a sequentially correct start order.

*Note:*When starting the application tree the start phase does not descend level by level but follows a branch of the tree (starting applications as it descends) before moving to the next branch.

There is an example of the recursive use of the application starter below and a diagram of the go start phase flow.

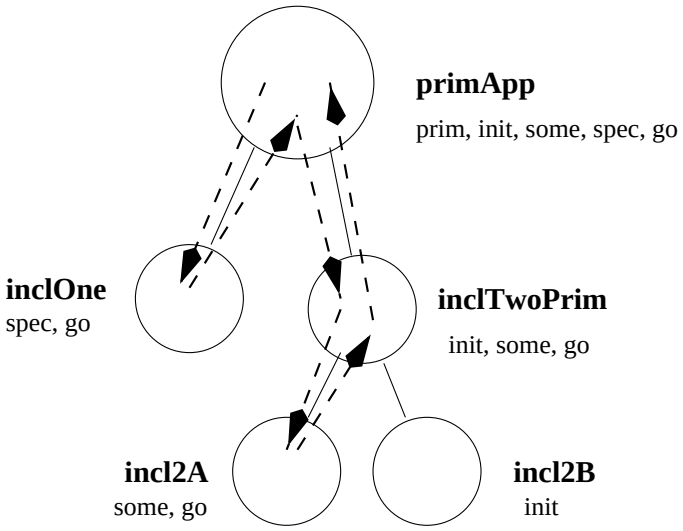


Figure 1.9: Flow of the go start phase.

primApp.app

```
:
{mod, {application_starter, [primApp, PrimAppStartArgs]},
{included_applications, [inclOne, inclTwoPrim]},
{start_phases, [{prim, PrimArgs}, {init, InitArgs}, {some, SomeArgs},
                {spec, SpecArgs}, {go, GoArgs}]},
:
```

inclOne.app

```
:
{mod, {inclOne, NotUsedArgs},
{included_applications, []},
{start_phases, [{spec, SpecArgs}, {go, GoArgsOne}]},
:
```

inclTwoPrim.app

```
:
{mod, {application_starter, [inclTwoPrim, NotUsedArgs]},
{included_applications, [incl2A, incl2B]},
{start_phases, [{init, []}, {some, []}, {go, []}]},
:
```

incl2A.app

```
:
{mod, {incl2A, []},
{included_applications, []},
{start_phases, [{some, SomeArgs2A}, {go, GoArgs2A}]},
:
```

incl2B.app

```
:
{mod, {incl2B, []},
{included_applications, []},
{start_phases, [{init, InitArgs2B}]},
:
```

evaluated functions

```
primApp:start(Type, PrimAppStartArgs)        % start the main supervisor
                                             % and permanent children
primApp:start_phase(prim, Type, PrimArgs)
primApp:start_phase(init, Type, InitArgs)
inclTwoPrim:start_phase(init, Type, [])
incl2B:start_phase(init, Type, InitArgs2B)
primApp:start_phase(some, Type, SomeArgs)
inclTwoPrim:start_phase(some, Type, [])
incl2A:start_phase(some, Type, SomeArgs2A)
primApp:start_phase(spec, Type, SpecArgs)
inclOne:start_phase(spec, Type, SpecArgs)
primApp:start_phase(go, Type, GoArgs)
inclOne:start_phase(go, Type, GoArgsOne)
inclTwoPrim:start_phase(go, Type, [])
incl2A:start_phase(go, Type, GoArgs2A)
```

Mixing applications with `start_phases` keys together with applications which have no `start_phases` keys is not allowed. The wrapper application's only function is to start an application which has no `start_phases` key. Without a wrapper the application will not start.

primApp.app

```
:
{mod, {primApp, PrimAppStartArgs},
{included_applications, [with, wrapper]},
{start_phases, [{init, InitArgs}, {go, GoArgs}]},
:
```

with.app

```
:
```

```
{mod, {with, NotUsedArgs},
{included_applications, []},
{start_phases, [{init, InitArgsW}, {go, GoArgsW}]},
:

wrapper.app
:
{mod, {wrapper, NotUsedArgs},
{included_applications, []},
{start_phases, [{init, WrapArgs}]},
:


evaluated functions
primApp:start(Type, PrimAppStartArgs)       % start the main supervisor
                                            % and permanent children
primApp:start_phase(init, Type, InitArgs)   % start the primary application
                                            % in the init phase
with:start_phase(init, Type, InitArgsW)     % start the with application
                                            % in init phase
wrapper:start_phase(init, Type, WrapArgs)   % start the without
                                            % application
primApp:start_phase(go, Type, GoArgs)       % start the primary application
                                            % in the go phase
with:start_phase(go, Type, GoArgsW)         % start the with application
                                            % in go phase
```

An Example

The following example assumes some familiarity with the Erlang boot script The example shows a simple boot script and config file for nodes cp1, cp2, cp3 and cp4, and applications myapp and localappl. These are the nodes and applications which are described in the previous section, with the exception that node cp4 has been added. This node only executes the local application localapp.

Boot Script

The following boot script is used at all four nodes. Each node makes an application:start call for myapp, although this distributed application is started only at one node.

```
{script,{"Dist Test","1.0"},
 [{preLoaded, [init,erl_prim_loader]},
  {progress, preloaded},
  <...>
  {progress,init_kernel_started},
  {apply,{application,load,
          [{application,
            myapp,
            [{description,"MYAPP"},
             {vsn,"1.0"},
```

```
            {modules, []},
            {applications,[kernel, stdlib]},
            {env,[]}]}]}},
 {apply,{application,load,
         [{application,
           localapp,
           [{description,"LOCALAPP"},
            {vsn,"1.0"},
            {modules, []},
            {applications,[kernel, stdlib, myapp]},
            {env,[]}]}]}},
 {progress,applications_loaded},
 {apply,{application,start,[kernel,permanent]}},
 {apply,{application,start,[stdlib,permanent]}},
 {apply,{application,start,[myapp,permanent]}},
 {apply,{application,start,[localapp,permanent]}},
 {progress,started}]}.
```

Config File

The following configuration file is used at all four nodes.

```
[{kernel, [{sync_nodes_optional, [cp1@cave, cp2@cave, cp3@cave, cp4@cave]},
{sync_nodes_timeout, 10000},
{distributed, [{myapp, [cp1@cave, {cp2@cave, cp3@cave}]}]}]}].
```

The application controller notices that `cp4` cannot start `myapp` and then ensures that `cp4` cannot start `localapp` until another node has started `myapp`.

1.3 Supervision Principles

This section introduces a process structuring model for programming fault tolerant applications. The model is based on the idea of *workers*, *supervisors*, and a *supervision tree*.

- Workers are processes which perform computations.
- Supervisors are processes which monitor the behaviour of workers. A supervisor can restart a worker if something goes wrong.
- The supervision tree is a hierarchical arrangement of an application into workers and supervisors.

A supervisor can supervise either workers or supervisors. Hereafter we will use the term *child* to mean either a worker or a supervisor.

Supervision trees can be manipulated by using the functions exported from the module `supervisor`.

A typical supervision tree looks as follows:

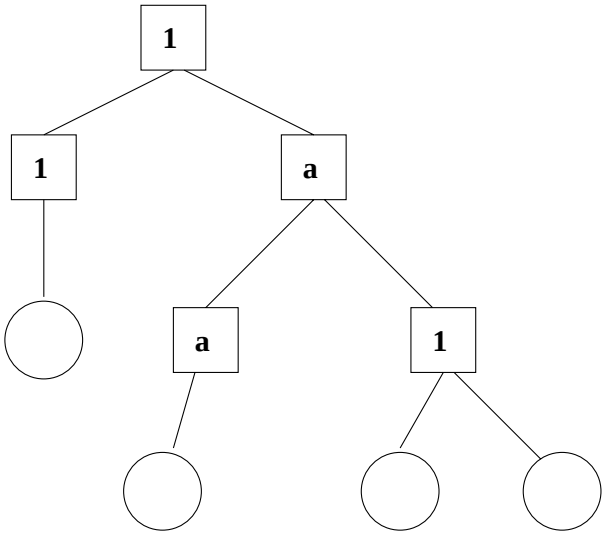


Figure 1.10: Supervision Tree

- The square boxes represent *supervisors*.
- The circles represent *workers*.
- The supervisors are annotated with either “1” (one_for_one), or “a” (one_for_all). This denotes the type of supervision strategy used.

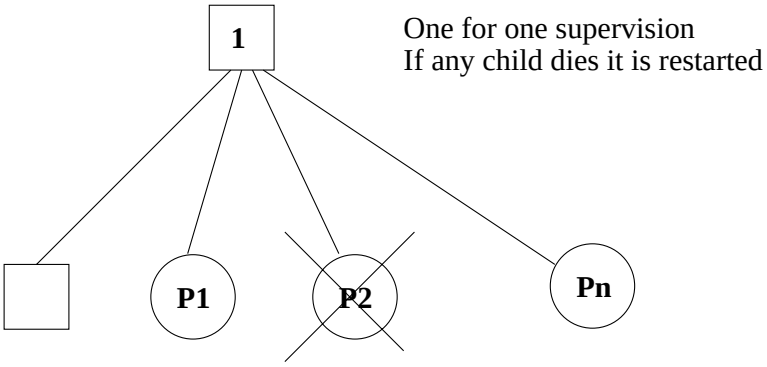


Figure 1.11: One_for_one Supervision

One_for_one Supervision. If a the supervised child dies, then this child only is restarted according to the start specification.

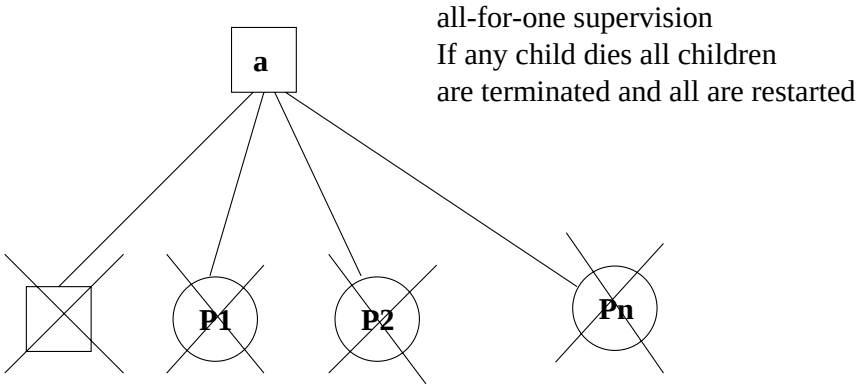


Figure 1.12: All_for_one Supervision

One_for_all Supervision. If a supervised child dies, then all the supervised children will be terminated according to the shutdown specification and all will then be restarted according to the start specification.

One_for_one supervision is often used when the processes supervised are *independent* of each other. If the processes are independent, they can often be programmed as sets of processes with no internal communication between them.

One_for_all supervision is often used when the processes supervised *depend* on each other. If the processes are dependent, they can often be programmed as sets of synchronized, communicating processes.

The Supervision Tree

Supervision trees are created by using an instance of the `supervisor` behaviour. A supervision tree is created from a supervision start specification. The module `xx` is an example of the supervision

behaviour. `xx:init(Args)` must be written so as to return {`ok, SuperSpec`}, or {`error, What`}, or `ignore`. The supervision tree is created with the following call:

```
supervisor:start_link(Name, xx, Args)
```

`Name` is the name of the supervision tree and can be used to manipulate the supervision tree. `Args` is an argument which is passed to `xx:init`. `Name` can be omitted if an anonymous supervisor is started.

```
-module(xx).
-behaviour(supervisor).
-export([init/1]).
init(_) ->
    {ok,{{one_for_all, MaxR, MaxT},
         [
          {log,   {log, start, [25]}, permanent, 5000,     worker,     [log]},
          {mysup, {mysup, start, []}, permanent, infinity, supervisor, [mysup]},
         ...
          {ChildName, MFA,            Type,      Shutdown, Class,      Modules}
        ]}}.
```

- `xx:init` returns a tuple of the form {`ok, SupSpec`}.
- `SupSpec` is a term of the form {`SupFlags, [ChildSpec]`}.
- `SupFlags` specifies the supervision to be of type one_for_one or one_for_all, and it specifies the restart frequency.
- `ChildSpec` is a list of specifications (one per child) which specifies how each child should be started and treated.

Each child can be one of three types:

- a *permanent* child is always restarted when it dies
- a *transient* child is restarted if it dies abnormally
- a *temporary* child is never restarted.

All child processes which are added to the supervision tree must implement the shutdown protocol correctly (refer to the section The Shutdown Protocol [page 24]). All the generic servers, event handlers, and other standard system behaviours automatically implement this protocol.

User processes which are implemented without using the standard behaviours must follow the shutdown protocol. This protocol can either be coded directly into the process, or it can be programmed using a supervisor bridge. Refer to the section Supervisor Bridge [page 26] in this chapter.

Refer also to the Reference Manual, the module `supervisor` in `stdlib`.

Restart of Processes in the Supervision Tree

If a one_for_all supervisor detects that one of its children has died, then the supervisor will terminate and restart all the processes it is supervising.

If a one_for_one supervisor detects that one its children has died, then the supervisor will terminate and restart only the process that died. However, if the supervisor itself terminates, it terminates all the children it supervises.

The restart behaviour will occur if the following conditions apply:

- the child process was a permanent process which dies for any reason; or
- the child process was a transient process which dies with abnormal exit; and
- the restart limit has not been exceeded.

Note:
A temporary process which dies will not cause any restart behaviour.

The Shutdown Protocol

The shutdown protocol is used when the supervisor decides to kill a child for any reason. This is achieved by evaluating `shutdown(Pid, Shutdown)`, where `Pid` is the Pid of the child, and `Shutdown` is the termination timeout value supplied in the start specification. `Pid` must be linked to the supervisor.

The supervisor process sends an exit signal to the child process and waits for acknowledgment. The child process should handle the `{'EXIT',ParentPid,shutdown}` message and terminate with reason `shutdown` if it traps exit signals. If an acknowledgment is not received within the specified time, then the child is killed.

Alternatively, the child can be terminated by specifying `Shutdown = infinity`, or `Shutdown = brutal_kill`. The exact behaviour is as follows:

```
shutdown(Pid, brutal_kill) ->
    exit(Pid, kill);
shutdown(Pid, infinity) ->
    exit(Pid, shutdown),
    receive
        {'EXIT', Pid, shutdown} -> true
    end;
shutdown(Pid, Time) ->
    exit(Pid, shutdown),
    receive
        {'EXIT', Pid, shutdown} ->
            true
    after Time ->
        exit(Pid, kill)
    end.
```

When a supervisor has to terminate several children, it terminates them synchronously in the reverse order of creation.

The Restart Frequency Limit Mechanism

The supervisors have a built-in mechanism to limit the number of restarts which can occur in a given time interval. This is determined by the values of the two parameters `MaxR` and `MaxT`. If more than `MaxR` number of restarts occur in the last `MaxT` seconds, then the supervisor shuts down all the children it supervises and dies.

If a supervisor terminates, then the next higher level supervisor takes some action. It either restarts the lower level supervisor or dies.

The intention of the restart mechanism is to prevent a situation where a process repeatedly dies for the same reason, only to be restarted again.

After the pre-determined number of restarts, the supervisor itself will die and the next higher level supervisor tries to correct the error by applying its restart strategy, and so on.

If a particular level in the system is incapable of correcting a given error, then it will eventually give up trying and pass the responsibility higher up in the supervision tree.

At some level in this process we hope that the error will finally be corrected.

Interpretation of the Restart Frequency

Assume that a number of restarts are performed at times:

```
T(0), T(1), T(2), ...
```

We define the quantity `Acc(N)` as follows:

```
Acc(N) = 1 + Acc(N-1) * (1 - Dt(N)/MaxT) when Dt(N) <= MaxR
Acc(N) = 1                                Otherwise
```

In the above expression, `Dt(N) = T(N) - T(N-1)`.

`Acc` represents an accumulator which increments by one every time a new restart is performed. The value of the accumulator decays linearly to zero in the time period `MaxT`, which is the measurement period.

If `Acc` is greater than the threshold value `MaxR`, then the supervising process dies.

Dynamic Processes in the Supervision Tree

In addition to the static supervision tree, we can also add dynamic children. A new child is added to an existing supervision tree with the following call:

```
supervisor:start_child(SupName, StartSpec) -> Result
```

SupName is the name,or the process identity, of the supervisor. StartSpec is a 6-tuple which contains the start specification of the child process, including the name of the child.

Children which are added using start_child/2 behave in the same manner as children defined in the start specification of the supervisor, with the following important exception:

> Warning:
> If a supervisor dies and is re-created, then all children which were dynamically added to the supervisor will be lost.

Any child, static or dynamic, can be stopped with the following call, which also stops the child in accordance with the shutdown specification for the child.

```
supervisor:terminate_child(Sup, ChildName)
```

A stopped child can be restarted with the following call:

```
supervisor:restart_child(Sup, ChildName)
```

A stopped child is permanently deleted with the following call, and cannot be restarted again:

```
supervisor:delete_child(Sup, ChildName)
```

As with start_child/2, any effects of dynamically adding or deleting children is lost if the supervisor itself restarts. In this case, the original start specification is used to restart the children.

The Supervisor Bridge

It can sometimes be useful to connect a process or a subsystem to the supervision tree, although it has not been designed with the supervision principles in mind. This can be accomplished by using an instance of the supervisor_bridge behaviour. A supervisor bridge is a process which sits in between a supervisor and the subsystem. It behaves like a real supervisor to its supervisor, but has a different interface than a real supervisor to its subsystem.

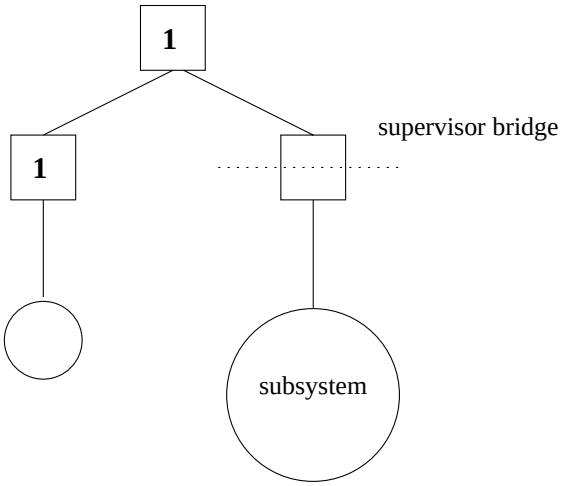


Figure 1.13: Supervision Bridge

The subsystem must choose one main process to be responsible for the subsystem, which the supervisor bridge will monitor. If this process terminates, the supervisor bridge will assume that the entire subsystem has terminated.

The `supervisor_bridge` behaviour must export two functions:

- `init/1`, which starts the subsystem and returns `{ok, Pid, State}`. `Pid` is the Pid of the main subsystem process, and `State` is any term.
- `terminate/2`, which is responsible for terminating the entire subsystem.

Suppose that we want to connect the `vshlr` server to a supervision tree. This server is described in the next section which is titled Servers [page 29]. If we use `vshlr_2` version of the server, we can connect it directly to a supervisor as it is written, since it uses the `gen_server` behaviour. If we use `vshlr_1` version instead, we must write a supervisor bridge to connect the server to a supervisor.

```
-module(vshlr_sup).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-behaviour(supervisor_bridge).

-export([start_link/0]).
-export([init/1, terminate/2]).

%% Creates a supervisor bridge for vshlr_1
start_link() ->
    supervisor_bridge:start_link({local, vshlr_sup}, vshlr_sup, []).

%% This function is supposed to start the vshlr server
init([]) ->
    vshlr_1:start(),
    case whereis(vshlr_1) of
        Pid when pid(Pid) -> {ok, Pid, []};
```

```
        undefined -> {error, start_error}
    end.

%% This function is supposed to stop the vshlr server
terminate(_Reason, []) ->
    vshlr_1:stop().
```

A supervisor child specification for this supervisor bridge looks as follows:

```
{vshlr_sup, {vshlr_sup, start_link, []}, permanent, 2000, worker, []}
```

C Code and Supervision

Parts of a system are likely to be written in C. This may be for perfomance reasons, or as a result of sourcing. While these C programs will never enjoy the supervision features available in Erlang, it is still possible to make these programs behave as if they were supervised Erlang processes. This is done by wrapping the C program with an Erlang stub, and this Erlang stub can be included as a worker in a supervision tree. The stub also relays supervision messages to the C program. The C program reacts to them in the form of “hooks”, instead of messages.

The supervision features is included in the `interface generator` module. Using this module, C programs with their Erlang interfaces can benefit from being supervised.

Refer to the chapter titled Interface Libraries, the section *The C Interface Generator* for a detailed explanation of C code supervision.

1.4 Servers

This section describes a simple and powerful way of programming client-server applications. Client-server applications are programmed using the `gen_server` behaviour.

Refer to the Reference Manual , the module `gen_server` in `stdlib`, for full details of the behaviour interface.

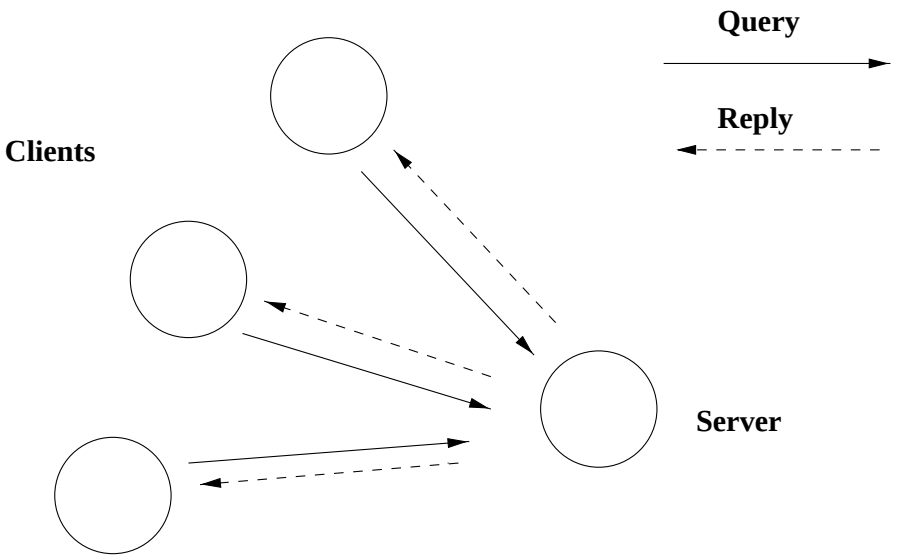
Client-Server Principles

This section describes several solutions to one sample problem in order to illustrate how to write client-server applications.

The sample problem is a very simple server which acts as a Home Location Register (HLR). We will implement a small sub-set of an HLR which we call VSHLR (Very Simple HLR) in a number of different ways. The Erlang modules which implement our VSHLR will always be called something like `vshlr_XX`. All these modules will export the following functions:

- `vshlr_XX:start() -> true` starts the server.
- `vshlr_XX:stop() -> true` stops the server.
- `vshlr_XX:i_am_at(Person, Position) -> ok` tells the server that `Person` is at the location `Position`.
- `vshrl_XX:find(Person) -> {at, Position} | lost` asks the server where `Person` is. The server responds `{at, Position}`, where `Position` is the last reported location, or `lost` if it does not know where the person is.

The client-server model can be illustrated in the following figure:



The Client-server model

Figure 1.14: The Client-Server Model

The client-server model is characterized by a central server and an arbitrary number of clients. The client-server model is generally used for resource management operations, where several different clients want to share a common resource. The server is responsible for managing this resource.

If we ignore how the server is started and stopped, and ignore all error cases, then it is possible to describe the server by means of a simple function `f`.

Suppose that the internal state of the server is described by the state variable `S` and that the server receives a query `Q`. The server responds by sending a reply `R` back to the client and changes its internal state to `S'`. This can be described as follows:

```
{R, S'} = f(Q, S)
```

Given a function `f`, we can write a very simple universal client server as follows:

```
-module(server).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-export([start/3, stop/1, loop/3, call/2]).

start(Name, F, State) ->
    register(Name, spawn(server, loop, [Name, F, State])).

stop(Name) ->
    exit(whereis(Name), kill).

call(Name, Query) ->
```

```
    Name ! {self(), Query},
    receive
        {Name, Reply} ->
            Reply
    end.

loop(Name, F, State) ->
    receive
        {Pid, Query} ->
            {Reply, State1} = F(Query, State),
            Pid ! {Name, Reply},
            loop(Name, F, State1)
    end.
```

vshlr can be written using server:

```
-module(vshlr_1).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-export([start/0, stop/0, i_am_at/2, find/1, handle_event/2]).

start() -> server:start(xx1,
                        fun(Event, State) ->
                            handle_event(Event, State)
                        end,
                        []).

stop() -> server:stop(xx1).

i_am_at(Person, Position) -> server:call(xx1, {i_am_at, Person, Position}).
find(Person)              -> server:call(xx1, {find, Person}).


handle_event({i_am_at, Person, Position}, State) ->
    State1 = update_position(Person, Position, State),
    {ok, State1};
handle_event({find, Person}, State) ->
    Location = lookup(Person, State),
    {Location, State}.

update_position(Key, Value, [{Key, _}|T]) -> [{Key, Value}|T];
update_position(Key, Value, [H|T])        -> [H|update_position(Key, Value, T)];
update_position(Key, Value, [])           -> [{Key,Value}].

lookup(Key, [{Key, Value}|_]) -> {at, Value};
lookup(Key, [_|T])            -> lookup(Key, T);
lookup(Key, [])               -> lost.
```

We can run this as follows:

```
1 > vshlr_1:start().
```

```
true
2> vshlr_1:i_am_at("joe", "home").
ok
3> vshlr_1:i_am_at("helen", "work").
ok
4> vshlr_1:find("joe").
{at,"home"}
5> vshlr_1:find("mike").
lost
6> vshlr_1:i_am_at("joe", {building,23}).
ok
7> vshlr_1:find("helen").
{at,"work"}
8> vshlr_1:find("joe").
{at,{building,23}}
```

Even though our VSHLR program is extremely simple, it illustrates and provides simple solutions to a surprisingly large number of design issues.

The reader should note the following:

- The functionality is divided between two different modules. *All* the code that deals with spawning processes and sending and receiving messages is contained in the module `server`. *All* the code that has to do with the *implementation* of the VSHLR is contained in the module `vshrl`. Note also that most of the functions in `vshrl` can be written in a pure, side effect free manner. This division of functionality is good programming practice.
- The code in `server` can be re-used to build many different client-server applications.
- The name of the server, in this example the atom `xx1`, is hidden from the users of the client functions. This means it can be changed without effecting the code that uses the client functions. This point has important consequences for writing distributed systems. Essentially, we can develop programs as non-distributed applications and then turn them into distributed applications by making very small changes to the client stub code. This point will be covered in more detail later.
- We hide the details of the remote procedure call inside the `server` module. This means that we can change how we do the remote procedure call at a later stage. This has consequences for error handling and the recovery from failures which may occur during a remote procedure call.
- We hide the details of the protocol used between the client and the server inside the `server` module. This is good programming practice and allows us to change the protocols without having to make any changes to the functions which use the server.

Extending the Server

Splitting a server into two parts means that we can work on either of the parts without effecting the other. We can illustrate this by extending the server so that it logs the last ten requests and calls the error logger if something goes wrong. This version is called `server1` to distinguish it from `server`.

```
-module(server1).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-export([start/3, stop/1, loop/4, call/2]).
```

```
start(Name, F, State) ->
    register(Name, spawn(server1, loop, [Name, F, State, []])).

stop(Name) ->
    exit(whereis(Name), kill).

call(Name, Query) ->
    Name ! {self(), Query},
    receive
      {Name, error} ->
        exit(server_error);
      {Name, {ok, Reply}} ->
        Reply
    end.

loop(Name, F, State, Buff) ->
    receive
      {Pid, Query} ->
        Buff1 = trim([Query|Buff]),
        case catch F(Query, State) of
          {'EXIT', Why} ->
            Pid ! {Name, error},
            error_logger:error_msg({server_error, Name, Buff1});
          {Reply, State1} ->
            Pid ! {Name, {ok, Reply}},
            loop(Name, F, State1, Buff1)
        end
    end.

trim([X1,X2,X3,X4,X5,X6,X7,X8,X9,X10|_]) -> [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10];
trim(X) -> X.
```

`server1` has exactly the same function interface as the previous version of `server`.

If we use `server1` together with `vshlr`, we get an improved version of `vshlr`, which has additional error handling facilities.

The improvement to VSHLR was made without any significant change to the code in the module `vshlr`. This is a consequence of dividing the server into two parts, the *generic* part which is common to all servers, and the *specific* part which concerns the VSHLR problem.

The Generic Server

The examples shown in the previous sections make it apparent that the server can be extended in a number of different ways. The module `gen_server` provides a number of useful extensions to our simple server. In the following example, `vshrl` is re-implemented using `gen_server`.

The Reference Manual, `stdlib`, module `gen_server` has detailed information about the generic server.

```
-module(vshlr_2).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-export([start/0, start_link/0, stop/0, i_am_at/2, find/1]).
```

```
-behaviour(gen_server).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2]).

%% These are the interface routines

start() -> gen_server:start({local, xx2}, vshlr_2, [], []).
start_link() -> gen_server:start_link({local, xx2}, vshlr_2, [], []).

stop()  -> gen_server:call(xx2, die, 10000).

i_am_at(Person, Location) ->
    gen_server:call(xx2, {i_am_at, Person, Location}, 10000).

find(Person)  ->
    gen_server:call(xx2, {find, Person}, 10000).

%% These Routine MUST be exported since they are called by gen_server

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

handle_call({i_am_at, Person, Location}, _, State) ->
    State1 = update_location(Person, Location, State),
    {reply, ok, State1};
handle_call({find, Person}, _, State) ->
    Location = lookup(Person, State),
    {reply, Location, State};
handle_call(die, _, State) ->
    %% ok goes back to the user and terminate(normal, State)
    %% will be called
    {stop,  normal, ok, State}.

handle_cast(Request, State) -> {noreply, State}.

handle_info(Request, State) -> {noreply, State}.

terminate(Reason, State) ->
    ok.

%% sub-functions

update_location(Key, Value, [{Key, _}|T]) -> [{Key, Value}|T];
update_location(Key, Value, [H|T])        -> [H|update_location(Key, Value, T)];
update_location(Key, Value, [])           -> [{Key,Value}].

lookup(Key, [{Key, Value}|_]) -> {at, Value};
lookup(Key, [_|T])            -> lookup(Key, T);
lookup(Key, [])               -> lost.
```

The flow of control in the example shown above is as follows:

- Start a new server by evaluation the expression `gen_server:start({local, xx2}, vshlr_2,`

`Args, Opts).`
This expression starts a *local* server with name `xx2` on the local node. The handler module `vshlr_2` is called to initialize the server.
The generic server calls `vshlr_2:init(Args)` which is expected to return `{ok, S}`. The value of `S` is used as the initial value of the state of the server.

- The client routines use the following type of calls:

  ```
  gen_server:call(xx2, {i_am_at, Person, Location}, 10000)
  ```

  For communication purposes, `xx2` is the name of the server and must agree with the name used to start the server. `{i_am_at, Person, Location}` is a command which is passed to the server, and 10000 is a timeout value. If the server does not respond within 10000 milliseconds, the call to the server is aborted.
  The previous call corresponds to the clause:

  ```
  handle_call({i_am_at, Person, Location}, _, State) ->
      State1 = update_Location(Person, Location, State),
      {reply, ok, State1};
  ```

  `handle_call` returns a tuple of the form`{reply, Reply, State1}`. In this tuple, `Reply` is the reply which should be sent back to the client, and `State1` is a new value for the state of the server.

- Stop the server by evaluating `gen_server:call(xx2, die, 10000).`
  This matches the following expression:

  ```
  handle_call(die, _, State) ->
      {stop, normal, ok, State}.
  ```

  The return value tells the server to stop. The server first evaluates `vshlr_2:terminate(normal, State)`. The reply, which is `ok` in this example, is passed back to the client and the server stops.

Local Client-Server

The example shown in the previous section was a local server. The main points to note were:

- `gen_server:start({local, xx2}, ...)` starts the server.
- `gen_server:call(xx2, ...)` calls the server.

Global Client-Server

To make a global server, the following small changes are made to the access routines:

- `gen_server:start({global, xx2}, ...)` starts the server.
- `gen_server:call({global, xx2}, ...)` calls the server.

With these changes, the client-server model will work in a network of distributed nodes. All nodes in the system are assumed to evaluate identical copies of the code. The server will be placed on the *first* node which evaluates `gen_server:start`. All other nodes will be coupled to this node automatically.

Anonymous Server

The following calls will start an anonymous server:

- `gen_server:start(Mod, ...) -> {ok, Pid}` starts an anonymous server. All calls to the server must include an explicit reference to the Pid of the server.
- `gen_server:call(Pid, ...)` calls the server.

The user must ensure that the Pid of the server is communicated to all clients which make use of the server.

Notes

- The server can be started with `gen_server:start`, or `gen_server:start_link`. In the case of `start_link`, the server is linked to the process which started the server.
- The `start_link` function must be used if the server is supervised by a supervisor.
- The server does not normally trap exits and will die if it receives an exit signal. If you wish the server to trap exits then you should evaluate `process_flag(trap_exit, true)` in `init/1` before returning `{ok, State}`.
- This section has only described the remote procedure call interface to the server. You can also send a *cast* to the server. In a cast, the client sends a message to the server and continues since no reply is sent by the server.
- The server can also handle exit messages and information requests from the management system. Refer to the Reference Manual, `stdlib` for more information.

1.5 Events

The event manager behaviour `gen_event` provides a general framework for building application specific event handling routines.

Refer to the Reference Manual , the module `gen_event` in `stdlib`, for full details of the behaviour interface.

Event managers provide named objects to which events can be sent. When an event arrives at an event manager, it will be processed by all the event handlers which have been installed within the event manager. None or several event handlers can be installed within a given event manager.

Event handlers can be written which act on all events in a particular class, on some of the events, or on some particular complex combination of events.

All events are processed by functions which are called from the module `gen_event` .

Event managers can be manipulated at runtime. In particular, we can install an event handler, remove an event handler, or replace one event handler with a different handler.

Event managers can be built for tasks like:

- error logging
- alarm handling
- call record logging
- debugging
- equipment management.

The event mechanism provides an extremely powerful model for building a large number of different applications. The following sections include examples of the kind of applications which can be built.

Definitions

The following definitions will help in understand this topic.

Event An occurrence, or something which happens.

Event Category The type or class of an event.

Event Manager A process which coordinates the processing of events of the same category.

Notification The act of informing an event manager that an event has occurred.

Event Handler A module which exports functions that can process events of a particular category. Event handlers can be installed within an event manager.

The Event Manager

The event manager essentially maintains a list of {Mod, State} pairs, which are called an MS list. For example:

```
[{Mod1, State1}, {Mod2, State2}, ...]
```

New modules are added to this list by calling gen_event:add_handler(EventManager, NewMod, Args).

EventManager is the name of the event manager, and NewMod is the name of an event handler and its callback module.

The event manager calls NewMod:init(Args), which is expected to return {ok, NewState}. If this happens, the tuple {NewMod, NewState} is added to the MS list.

When an application generates an event by calling gen_event:notify(EventManager, Event), the event Event is delivered to the event manager.

The event manager then processes the event by calling Mod:handle_event(Event, State) for each module in the MS list. This has the effect of replacing the MS list [{Mod1, State1}, {Mod2, State2}, ...] with [{Mod1, State1p}, {Mod2, State2p}, ...], where:

```
{ok, State1p} = Mod1:handle_event(Event, State1)
{ok, State2p} = Mod2:handle_event(Event, State2)
```

The event manager can be thought of as a generalization of a conventional finite state machine. Instead of a single state, we maintain a set of states, and a set of state transition functions.

We further generalize this mechanism by allowing handle_event to return not only a new state, but also by allowing it to request a change of the event handler, or to request the removal of the existing event handler. What happens is shown by the following pseudo-code example which executes within gen_event. The callback functions Mod1:terminate(...) and Mod2:init(...) must also be supplied by the user.

```
notify(Event, Mod1, State) ->
     case Mod1:handle_event(Event, State) of
          {ok, State1} ->
               ... add  {Mod1, State1} to the MS list
          remove_handler ->
               Mod1:terminate(remove_handler, State),
               ... delete the handler from the MS list
          {swap_handler, Args1, State1, Mod2, Args2}
               State2 = Mod1:terminate(Args1, State1),
               {ok, State2a} = Mod2:init({Args2, State2}),
               ... add {Mod2, State2a} to the MS list and delete
                                              the Mod1 handler
```

The handler returns the following values:

- {ok, State}. The new state is added to the MS list.
- remove_handler. The handler is finalized and then removed from the MS list.
- {swap_handler, ...}. The handler is finalized and the return value is passed into the init function of the new handler.

You can also send a request to a specific handler in the MS list by evaluating `gen_event:call(EventManager, Mod, Query)`, which returns the value obtained by evaluating `Mod:handle_call(Query, State)`.

You remove a handler with the call `gen_event:delete_handler(EventManager, Mod, Args)`, which returns the value obtained by evaluating `Mod:terminate(Args, State)`, where `State` is the state associated with `Mod` in the MS list.

Finalization

Each time a new handler is installed, `Mod:init(...)` is called, and each time a handler is removed `Mod:terminate(...)` is called.

The act of calling a specific routine every time a handler is removed is called "finalization". The finalization routine `terminate` has two arguments:

- `Args` which is the reason for termination
- `State` which is the current value of the state.

`Mod:terminate/2` is expected to return a new state. Depending on the context, this state is sometimes ignored and sometimes passed into a new initialization routine.

Writing an Event Manager

To create a new event manager, we evaluate the function `gen_event:start(Manager)`, where `Manager` is the name of the event manager.

For example, the call `gen_event:start({local, error_logger})` starts a new (local) event manager called `error_logger`. Note that calling `gen_event:start({local, Manager})` has the side effect of creating a new registered process named `Manager`.

> Note:
> We could also create a *global* event manager by calling `gen_event:start({global, event_logger})`.

So far, the error logger cannot do anything and we have to install a handler. The function `gen_event:add_handler(Manager, Handler, Args)` can be used to install the handler `Handler` in the event manager `Manager`.

When `gen_event:add_handler(Manager, Handler, Args)` is called, the event manager calls the function `Handler:init(Args)` which normally returns `{ok,State}`. The value of `State` is stored in the event manager together with the name of the handler.

Any process can send an event to the event manager by evaluating the function `gen_event:notify(Manager, Event)`. When this happens, the event manager processes the event by calling the function `Handler:handle_event(Event, State)`. This is done for each handler which has been installed in the manager. The function `Handler:handle_event(Event, State)` should return one of three different values:

- `{ok, State}`. `State` is a new state which will be used the next time the handler is called.

- `remove_handler`. This tells the event manager to remove this event handler by calling `Handler:terminate(remove_handler, State)`.
- {`swap_handler, Args1, State1, Mod2, Args2`}. This tells the event manager to remove the current event handler by calling `State2 = Handler:terminate(Args1, State1)`, and then to install a new handler by calling `Mod2:init(Args2, State2)`

An Error Logger

The module `error_logger_memory_h` provides a simple memory resident error logger. It stores at most `Max` error messages. After this all error messages are lost.

```
-module(error_logger_memory_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/roma/2 $ ').

-behaviour(gen_event).

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

init(Max) -> {ok, {Max, 0, []}}.

handle_event(Event, {1, Lost, Buff}) ->
    {ok, {1, Lost+1, Buff}};
handle_event(Event, {N, Lost, Buff}) ->
    {ok, {N-1, Lost, [{event1, date(), time(), Event}|Buff]}}.

handle_info(_, S) -> {ok, S}.

handle_call(_, S) -> {ok, ok, S}.

terminate(swap_to_file, {_, 0, Buff}) ->
    {error_logger_memory_h, Buff};
terminate(swap_to_file, {_, Lost, Buff}) ->
    {error_logger_memory_h,
      [{event1,date(),time(),{Lost, messages_lost}}|Buff]};
terminate(_, State) ->
    ... display the data using a secret internal BIF ...
    ...
```

To start a simple memory based error logger which can store at most 25 messages we evaluate:

```
gen_event:start({local, error_logger}),
gen_event:add_handler(error_logger, error_logger_memory_h, 25).
```

To log an error, an application evaluates the expression:

```
gen_event:notify(error_logger, Event)
```

This error logger is similar to the error logger installed in the system kernel when the system boots. Be aware that no file system has been installed just after the system has started, so if any errors occur they are stored in memory. This error logger is perfectly adequate for recording errors which occur when booting the system.

The simple error logger shown can be improved by doing something more intelligent with the errors.

- If a file system is installed, then we might want to store the errors in disk files.
- If we have no file system, but we are a distributed Erlang node, then we might wish to send the errors to some other node in the system where they can be stored.

The following example shows a handler which stores events on disk:

```
-module(error_logger_file_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/3 $').

-behaviour(gen_event).

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

init({{Fname,Max,N}, {error_logger_memory_h, Buff}}) ->
     {ok, {{Fname, N}, length(Buff), Max, Buff}}.

handle_event(Event, {F, N, Max, Buff}) ->
     Buff1 = [{event1, date(), time(), Event}|Buff],
     N1 = N + 1,
     if
         N1 > Max -> {ok, {dump_events(F, Buff1), 0, Max, []}};
         true     -> {ok, {F, N1, Max, Buff1}}
     end.

handle_info(_, S) -> {ok, S}.

handle_call(_, S) -> {ok, ok, S}.

terminate(_, {F, N, Max, Buff}) ->
    dump_events(F, Buff),
    ok.

dump_events(F, []) ->F;
dump_events({File, Index}, Buff) ->
    Fname = File ++ integer_to_list(Index) ++ ".log",
    file:write_file(Fname, term_to_binary(Buff)),
    {File, Index + 1}.
```

This handler has been explicitly written to take over from the simple error handler. To swap handlers so that all errors are logged on disk we can evaluate:

```
gen_event:swap_handler(error_logger,
                       {error_logger_memory_h, swap_to_file},
                        {file_error_handler_h, {"/usr/local/file/log",100,45}}).
```

Each disk file will contain 100 events. These files will be called `/usr/local/file/log45.log`, `/usr/local/file/log46.log`, and so on.

The reader should also examine this example carefully and observe the flow of control between the finalization routine in the memory resident error logger, and the initialization routine in the file logger.

An Alarm Handler

We start by creating an alarm manager.

```
gen_event:start({local, alarm}).
```

That is all you need to do to make an alarm manager. Any process can now generate an alarm by evaluating `gen_event:notify(alarm, Event)`.

For example, to say that apparatus one is overheating you might call:

```
gen_event:notify(alarm, {hardware, 1, overheating}).
```

This alarm is then delivered to the alarm manager. However, the alarm manager will ignore the alarm since no alarm handlers have been installed.

The following example shows how to write and install an alarm handler which sends all alarms to the error logger:

```
-module(log_all_alarms_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-behaviour(gen_event).

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

init(_) -> {ok, 1}.

handle_event(Event, N) ->
    gen_event:notify(error_logger, {alarm, N, Event}),
    {ok, N + 1}.

handle_info(_, S) -> {ok, S}.

handle_call(_,S) -> {ok, ok, S}.

terminate(_, _) -> ok.
```

The next example shows an alarm handler which is only interested in alarms from hardware1. This handler counts the alarms and stops the hardware if 10 alarms have arrived:

```
-module(hardware_1_alarms_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/roma/1 $').

-behaviour(gen_event).

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

init(_) -> {ok, 1}.

handle_event({hardware, 1, What}, N) ->
     N1 = N + 1,
```

```
     if
         N1 == 10 ->
             %% .... code to stop hardware 1 ....
             {ok, true};
         true ->
             {ok, N + 1}
     end;
handle_event(_,State) ->
    %% This catches all other events intended
    %% for other handlers
    {ok, State}.

handle_call(_,State) -> {ok, ok, State}.

terminate(_, _) -> ok.
```

Both of these alarm handlers are installed in the alarm manager as follows:

```
gen_event:add_handler(alarm, log_all_alarms_h, []),
gen_event:add_handler(alarm, hardware_1_alarm_h, []),
```

Both handlers will run concurrently. A specialized handler can be added and removed at any time. Note also the second clause of `handle_event`. Since our handler must succeed for any event we add a final “catch all” clause and make sure it returns the original state.

Exit Notification

This section describes how to monitor a process and send a message to the error logger if the process terminates with an abnormal exit.

```
-module(at_exit_log_error_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/roma/1 $ ').

-behaviour(gen_event).

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

init(_) ->
    process_flag(trap_exit, true),
    {ok, []}.

handle_event({monitor, Pid}, S) ->
    link(Pid), {ok, S};
handle_event(_, S) ->
    {ok, S}.

handle_info({'EXIT', _, normal}, S) ->
    {ok, S};
handle_info({'EXIT', Pid, Why}, S) ->
    gen_event:notify(error_logger, {non_normal_exit, Pid, Why}),
    {ok, S};
```

```
handle_info(_, S) -> {ok, S}.

handle_call(_, S) -> {ok, ok, S}.

terminate(_, _) -> ok.
```

To start the handler we evaluate:

```
gen_event:start({local, at_exit}),
gen_event:add_handler(at_exit, at_exit_log_error_h, []).
```

A monitoring process is started with `gen_event:notify(at_exit, {monitor, Pid}).` where `Pid` represents the process that we wish to monitor.

Exit Handler

This section describes how to trigger an event to occur when a process exits.

```
-module(at_exit_apply_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/roma/1 $ ').

-behaviour(gen_event).

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

init(_) ->
    process_flag(trap_exit, true),
    {ok, []}.

handle_event({at_exit_apply,Pid,MFA},S) -> {ok, [{Pid, MFA}|S]};
handle_event(_, S)                      -> {ok, S}.

handle_info({'EXIT', Pid, _}, S) -> {ok, do_exit_actions(Pid,S,[])};
handle_info(_, S) -> {ok, S}.

handle_call(_, S) -> {ok, ok, S}.

terminate(_, _) -> [].

do_exit_actions(Pid, [{Pid, {M,F,A}}|T], L) ->
    catch apply(M, F, A),
    do_exit_actions(Pid, T, L);
do_exit_actions(Pid, [H|T], L) ->
    do_exit_actions(Pid, T, [H|L]);
do_exit_actions(Pid, [], L) ->
    L.
```

Install the handler as follows:

```
gen_event:start({local, at_exit}).
gen_event:add_handler(at_exit, at_exit_apply_h, []).
```

Set an event as follows:

```
gen_event:notify(at_exit, {at_exit, Pid, MFA})
```

Now, whenever `Pid` dies, `MFA` will be applied.

One or Many Handlers

The previous sections describe three different `at_exit` handlers. When designing a system we have to decide whether to install three different handlers in the same manager, or to create three different managers each with a single handler.

The following two examples produce the same effect.

Example 1:

```
gen_event:start({local, at_exit}).
gen_event:add_handler(at_exit, at_exit_apply_h, []).
gen_event:add_handler(at_exit, at_exit_log_error_h, []).
...
gen_event:notify(at_exit, {monitor, Pid}).
gen_event:notify(at_exit, {at_exit_apply, Pid, MFA}).
```

Example 2:

```
gen_event:start({local, at_exit_apply}).
gen_event:add_handler(at_exit_apply, at_exit_apply_h, []).
gen_event:start({local, at_exit_log_error}).
gen_event:add_handler(at_exit_log_error, at_exit_log_error_h, []).
...
gen_event:notify(at_exit_apply, {monitor, Pid}).
gen_event:notify(at_exit_log_error, {at_exit_apply, Pid, MFA}).
```

The first example creates one manager and installs two handlers. The second example creates two managers each with a single handler.

The first strategy is more flexible and will allow more handlers to be added at runtime, but at the cost of reducing concurrency in the system.

Plug and Play

This example assumes that a number of hardware drivers have been written which can automatically detect when hardware is added or removed from a system. The following functions are used to add and remove hardware from the system:

- `gen_event:notify(plug_and_play, {added, Hw})`. Use this function to add hardware.
- `gen_event:notify(plug_and_play, {removed, Hw})`. Use this function to remove hardware.

The handler `plug_and_play_db_h` maintains a database of all plug and play hardware which has been added to the system:

```
-module(plug_and_play_db_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/roma/1 $').

-behaviour(gen_event).

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

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

handle_event({added, Hw}, S)   -> {ok, [Hw|S]};
handle_event({removed, Hw}, S) -> {ok, lists:delete(Hw, S)};
handle_event(_, S)             -> {ok, S}.

handle_info(_, S) -> {ok, S}.

handle_call(what_hardware, State) -> {ok, State, State}.

terminate(_, _) -> ok.
```

This code just keeps a record of all hardware that has been started in a list. You can ask what hardware has been installed by evaluating the following function, which returns a list of the hardware that the plug and play manager knows about.

```
gen_event:call(plug_and_play, plug_and_play_db_h, what_hardware)
```

The following example shows a specialized handler which serves the purpose of doing something special when a piece of hardware is added, and doing something different when this piece of hardware is removed. The example is written for a sound card:

```
-module(plug_and_play_sound_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-behaviour(gen_event).

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

init(_) -> {ok, none}.

handle_event({added, {soundcard, X}}, S) ->
```

```
    Pid = soundcard:start(X),
    {ok, [{X, Pid}|S]};
handle_event({removed, {soundcard, X}}, S) ->
    {ok, stop_card(X, S, [])};
handle_event(_, S) ->
    {ok, S}.

stop_card(X, [{X, Pid}|T], L) -> soundcard:stop(Pid), lists:reverse(L, T);
stop_card(X, [H|T], L)        -> stop_card(X, T, [H|L]);
stop_card(X, [], L)           -> L.

handle_info(_, S) -> {ok, S}.

handle_call(_,S) -> {ok, ok, S}.

terminate(_,_) -> ok.
```

The plug-and-play manager can take care of both the plug-and-play database and the special processing of sound cards, when they are added and removed.

```
gen_event:start({local, plug_and_play}).
gen_event:add_handler(plug_and_play, plug_and_play_db_h, []).
gen_event:add_handler(plug_and_play, plug_and_play_sound_h, []).
```

Trace Logger

This section describes a simple handler which can trace all “foo” events.

```
-module(trace_foo_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/roma/1 $').

-behaviour(gen_event).

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

init(File) ->
    {ok, Stream} = file:open(File, write),
    {ok, Stream}.

handle_event({foo, X}, F) -> io:format(F, "~w~n", [X]), {ok, F};
handle_event(_,S)         -> {ok, S}.

handle_info(_, S) -> {ok, S}.

handle_call(_,S) -> {ok, ok, S}.

terminate(_, S) -> file:close(S), ok.
```

If you start the tracer with the function `gen_event:start({local, tracer}).` and trace “foo” events with the call `gen_event:notify(tracer, {foo, ...}).`, nothing will happen.

If you install a trace handler by calling `gen_event:add_handler(tracer, trace_foo_h, "/usr/local/file1").`, then all foo events will be written to the file "`/usr/local/file1`".

Evaluating `gen_event:remove_handler(tracer, trace_foo_h).` removes the handler and closes the file at the same time .

> Note:
> This example supplies arguments to both `init` and `terminate`.

Encapsulation

In all the examples shown in this section, `gen_event` function calls have been used instead of encapsulating the different functions which access the event manager. In the following example, the interface routines `start/0`, `stop/0`, `added/1`, `removed/1` and `which/0` are added to the code for `plug_and_play.erl`.

```
-module(plug_and_play).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-behaviour(gen_event).

-export([start/0, stop/0, added/1, removed/1, which/0]).
-export([init/1, handle_event/2, handle_call/2, terminate/2]).

start() ->
    gen_event:start({local, plug_and_play}),
    gen_event:add_handler(plug_and_play, plug_and_play, []).

stop()      -> gen_event:stop(plug_and_play).
added(Hw)   -> gen_event:notify(plug_and_play, {added, Hw}).
removed(Hw) -> gen_event:notify(plug_and_play, {removed, Hw}).
which()     -> gen_event:call(plug_and_play, plug_and_play, what_hardware).

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

handle_event({added, Hw}, S)      -> {state, [Hw|S]};
handle_event({removed_hw, Hw}, S) -> {state, lists:delete(Hw, S)};
handle_event(_, S)                -> {state, S}.

handle_call(what_hardware, State) -> {ok, State, State}.

terminate(_, _) -> ok.
```

This module should now be accessed through its interface routines only, and all details on how it was implemented using `gen_event` can be omitted.

1.6 Finite State Machines

This section describes the general principles of the Finite State Machine (FSM) behaviour and shows how to make FSM based applications. Refer to the Reference Manual `stdlib`, the module `gen_fsm` for additional information about this behaviour.

Many applications can be modeled as FSMs and be programmed using the `gen_fsm` behaviour. Protocol stacks are such an example.

A FSM can be described as a set of relations of the form:

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

These relations are interpreted as meaning:

> If we are in state `S` and the event `E` occurs, we should perform the actions `A` and make a transition to the state `S'`.

If you program an FSM using the `gen_fsm` behaviour, then the state transition rules should be written as a number of Erlang functions which conform to the following convention:

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

The figure below is a simple FSM describing “Plain Ordinary Telephony Service” (POTS).

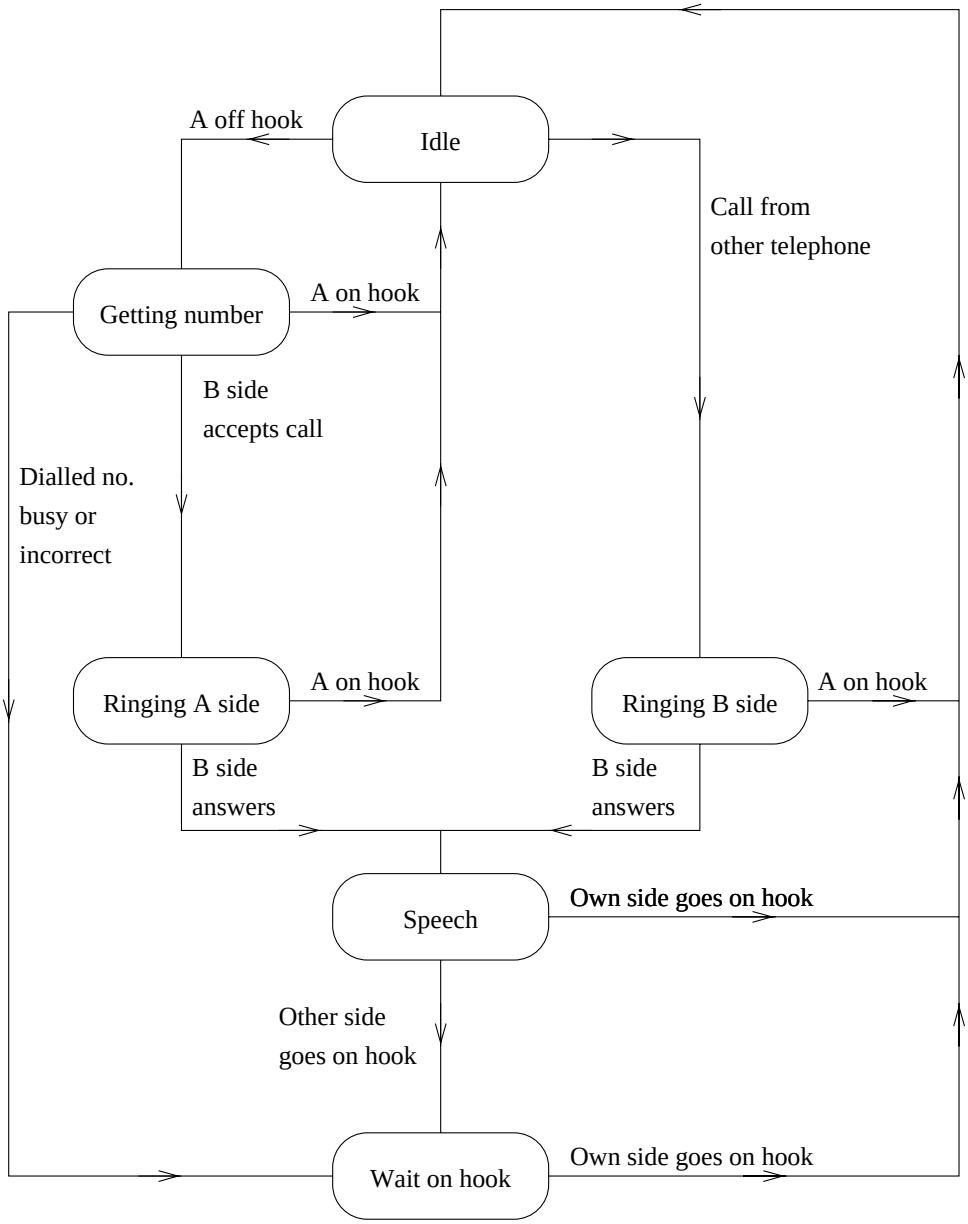


Figure 1.15: FSM Example

The POTS FSM can be described by the following `gen_fsm` behaviour:

```
init(A) ->
    {ok, idle, A}.

idle({off_hook, A}, A) ->
    {next_state, getting_number, {A,[]}};
idle({seize, A}, B) when A /= B ->
    {next_state, ringing_b_side, {B, A});
```

```
idle(_, A) ->
    {next_state, idle, A}.

getting_number({digit, D}, {A, Seq}) ->
    case ... of
        ... ->
            ...
                {next_state, ringing_a_side, {A, B}};
        ... ->
            ...
                {next_state, getting_number, {A, Seq1}};
        ... ->
            ...
                {next_state, wait_on_hook, A}
    end;
getting_number({on_hook, A}, {A,_}) ->
    {next_state, idle, A}.

ringing_a_side({on_hook, A}, {A, B}) ->
    {next_state, idle, A};
ringing_a_side({answered, B}, {A, B}) ->
    {next_state, speech, {A,B}}.

ringing_b_side({on_hook, A}, {B, A}) ->
    {next_state, idle, B};
ringing_b_side({off_hook, B}, {B, A}) ->
    {next_state, speech, {B, A}}.

speech({on_hook, A}, {A, B}) ->
    {next_state, idle, A};
speech({on_hook, B}, {A, B}) ->
    {next_state, wait_on_hook, A}.

wait_on_hook({on_hook, A}, A) ->
    {next_state, idle, A}.
```

The code shown above only describes the state transitions. To add the actions, we might add the following code:

```
getting_number({digit, D}, {A, Seq}) ->
    Seq1 = Seq ++ [D],
    case number_analyser:analyse(Seq1) of
        {user, B} ->
            hw:seize(B, A),
            {next_state, ringing_a_side, {A, B}};
        get_more_digits ->
            {next_state, getting_number, {A, Seq1}};
        invalid_number ->
            hw:send_nasty_tone(A, bad_number_tone),
            {next_state, wait_on_hook, A}
    end;
getting_number({on_hook, A}, {A,_}) ->
    hw:stop_codec(A),
```

```
    {next_state, idle, A}.
```

To complete this example, we have to package the FSM and the event generation routines in a behaviour module, as shown in the following example, and then add the code for the FSM.

```
-module(pots).
-behaviour(gen_fsm).
-export([...]).

start() -> gen_fsm:start(...)

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

on_hook(A) -> gen_fsm:send_event(..., {off_hook, A}).
...
```

An FSM Example

The simple POTS example shown above does not include all required details. The following example is complete and hopefully also self explanatory.

```
-module(test1_fsm).
-behaviour(gen_fsm).

%% interface routines

%% start us up
start() -> gen_fsm:start({local, hello}, test1_fsm, [], []).

%% send a hello -- this will end up in the FSM routines
hello() -> gen_fsm:send_event(hello, {hello, self()}).

%% send a stop this will end up in "handle_event"
stop()  -> gen_fsm:send_all_state_event(hello, stopit).

%% -- interface end

%% This initialisation routine gets called
init(_) ->
    {ok, idle, []}.

%% The state machine
idle({hello, A}, []) ->
    {next_state, one, A}.

one({hello, B}, A) ->
    A ! {hello, B},
    B ! {hello, A},
    {next_state, idle, []}.
```

```
%% this handles events in *any* state
handle_event(stopit, AnyState, TheStateData) ->
    {stop, i_shall_stop, []}.   %% tell it to stop


%% This is the finalisation routine - it gets called
%% When we have to stop
terminate(i_shall_stop, TheStateIwasIn, TheStateData) ->
    ok.
```

Other Ways of Programming FSMs

This section has focused on the `gen_fsm` behaviour. A very large FSM, or an FSM which has a very regular structure, can be built another way. It is possible to automatically generate the code which describes the machine from the FSM specification.

1.7 Special Processes

This section describes how to write a process which understands and behaves like the generic processes, for example gen_server and gen_fsm. In this context, behaves means:

- the process takes care of special system messages
- it creates a crash report if terminated abnormally.

All processes should be started in a supervision tree and they must respond to system messages when started this way.

System messages are used to change code in a controlled way and they are synchronized by a dedicated process which is called the release handler. Other typical system messages are requests for process status, and requests to suspend or resume process execution and debug messages.

Starting a Process

The proc_lib module should be used to start a process. This process wraps the initial function call with a catch, and a crash report is generated if the process terminates with another reason than normal or shutdown.

The function which starts a new process shall always return {ok,Pid} when successfully started, or {error,Reason} in case of failure. One of the proc_lib:start_link or spawn_link functions must be used when the process is included in a supervision tree . The following simple example illustrates this:

```
-module(test).

-export([start/0,init/1]).

start() ->
    case whereis(test_server) of
        undefined ->
            Pid = proc_lib:spawn_link(test, init, [self()]),
            {ok, Pid};
        Pid ->
            {error, {already_started, Pid}}
    end.

init(Parent) ->
                register(test_server, self( )),
    %% here is the new process.
```

System Messages

System messages are received as: {system, From, Request}. The content and meaning of this message are not interpreted by the receiving process module. When a system message has been received the function sys:handle_system_msg(Request, From, Parent, Mod, Deb, Misc) is called in order to handle the request. The arguments of this function have the following meaning:

- Request is any term
- From is the process identity of the calling process
- Parent is the parent process identity
- Mod is the current module
- Deb is a list of debug information
- Misc is any term describing the internal state.

> Note:
> The handle_system_msg/6 function never returns. It calls one of the functions system_continue/3 or system_terminate/4 to return to the original module. These functions are described in the following list.

The Mod module must export the following functions, which may be called from the sys:handle_system_msg/6 function:

- system_continue(Parent, Deb, Misc), where Misc is the internal state (i.e. loop data) forwarded in the above call to handle_system_msg/6. This function resumes normal execution. .
- system_terminate(Reason, Parent, Deb, Misc). The Parent process has terminated with Reason, or ordered us to terminate according to the shutdown protocol [page 24]. This provides a chance to clean up before terminating.
- system_code_change(Misc, OldVsn, Module, Extra) -> {ok, NMisc} | Error. In this case, our process has been ordered to perform a code change. Extra gives extra information about the code change. OldVsn is the old version of Module. The system_code_change function is executed in the newly loaded version of the module. This function should return the internal state, possibly modified in order to fulfil the needs of the new module.

> Note:
> The version is specified as an attribute -vsn(Vsn). in the Erlang source code.

According to the shutdown protocol [page 24], a {'EXIT',Parent,Reason} message from Parent is an order to terminate. Normally one shall terminate the process with the same Reason as Parent.

Other Messages

If the modules used to implement the process can change dynamically during runtime, there is one more message a process must understand. An example is the gen_event processes, which add new handlers at runtime.

This message is {get_modules, From}. The reply to this message is {modules, Modules}, where Modules is a list of the currently active modules in the process.

This message is used by the release handler to find which processes execute a certain module. A process may then be suspended and ordered to perform a code change for one of its modules.

Debugging

The module sys implements some standardized debug and trace facilities. The Deb information passed with the handle_system_msg function can be manipulated, created and inspected using the following functions:

- sys:debug_options([Opt]) -> Deb. This function creates the Deb information. Opt is one of trace | log | statistics | {log_to_file, FileName} | {install, {Func, FuncState}}.
- sys:get_debug(Opt, Deb, Default) -> Value. This function fetches the current value of Opt in Deb. Default is any term which describes the default value.
- sys:handle_debug(Deb, FormFunc, Info, Event) -> Deb. This function is called whenever we want to handle an event as a debug event. It has the following arguments:
  - FormFunc is one of {Module, Function} or a fun with arity 3. This function is called as FormFunc(Dev,Event,Info), where Dev is used in the same manner as in io:format(Dev,Format,Args). The FormFunc function is used to display the events.
  - Info is any term passed to FormFunc.
  - Event is {in, Msg} | {in, Msg, From} | {out, Msg, To} | term()

The following functions in the module sys can be used to activate or de-activate debugging, or to install your own trigger/debug functions: log/2, log/3, trace/2 trace/3, statistics/2, statistics/3, log_to_file/2, log_to_file/3, no_debug/1, no_debug/2, install/2, install/3, remove/2, remove/3. Refer to the Reference Manual, stdlib, module sys for details.

An Example

The following example of a simple server illustrates how to use the special processes described.

```
-module(test).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').

-export([start/1, init/2, system_continue/3,
         system_terminate/4, write_debug/3]).

start(Options) ->
    case whereis(test_server) of
        undefined ->
            Pid = proc_lib:spawn_link(test, init, [self(), Options]),
```

```
            register(test_server, Pid),
            {ok, Pid};
        Pid ->
            {error, {already_started, Pid}}
    end.

init(Parent, Options) ->
    process_flag(trap_exit, true),
    Deb = sys:debug_options(Options),
    loop([], Parent, Deb).

loop(State, Parent, Deb) ->
    receive
        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent, test, Deb, State);
        {'EXIT', Parent, Reason} ->
            cleanup(State),
            exit(Reason);
        {From, OurMsgs} ->
            NewDeb = sys:handle_debug(Deb, {test, write_debug},
                                      test_server, {in, OurMsgs, From}),
            {Answer, NewState} = do_something(OurMsgs, State),
            From ! {self(),Answer},
            NewerDeb = sys:handle_debug(NewDeb, {test, write_debug},
                                        test_server,
                                        {out, {self(), Answer}, From}),
            loop(NewState, Parent, NewerDeb);
        What ->
            NewDeb = sys:handle_debug(Deb, {test, write_debug},
                                      test_server, {in, What}),
            loop(State, Parent, NewDeb)
    end.

cleanup(State) ->
    ok.

do_something(Msg, State) ->
    %% Here we shall perform actions to handle the request.
    {ok, State}.

%% Here are the sys call back functions

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

system_terminate(Reason, Parent, Deb, State) ->
    cleanup(State),
    exit(Reason).

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

This can be used as follows:

```
1> test:start([trace]).
{ok,<0.21.0>}
2> test_server ! hej.
test_server event = {in,hej}
hej
3> test_server ! {self(), hopp}.
test_server event = {in,hopp,<0.18.0>}
{<0.18.0>,hopp}
test_server event = {out,{<0.21.0>,ok},<0.18.0>}
4> receive X -> X end.
{<0.21.0>,ok}
5> sys:trace(test_server, false).
ok
6> sys:log(test_server, true).
ok
7> test_server ! message_1.
message_1
8> test_server ! message_2.
message_2
9> sys:log(test_server, print).
test_server event = {in,message_1}
test_server event = {in,message_2}
ok
```

1.8 Writing an Application

This chapter describes and examplifies the following tasks, which all make up the larger task of writing an Erlang application:

- how to structure an application
- how to create a supervision tree
- how to use the common behaviours
- how to install event handlers
- how to configure an application
- how to write an application specification
- how to test an application
- how to write a distributed application.

Structuring the Application

Every application has an application master process that monitors the behaviour of the entire application. It starts the application by calling the start function specified in the application specification. This start function is assumed to start one process that is the main process of the application. Normally, this process is a supervisor, but it could also be a supervisor bridge.

Most applications are structured as a supervision tree, where the main process is the root supervisor of the application, and all other processes in the application are located somewhere under this supervisor.

To illustrate this point, suppose that we want to build an application named `hlr`. We want this application to contain the `vshlr` server introduced in Client-Server Principles [page 29], and a special alarm event handler. This simple application will then have one supervisor and one worker as shown in Illustration of HLR Application [page 60].

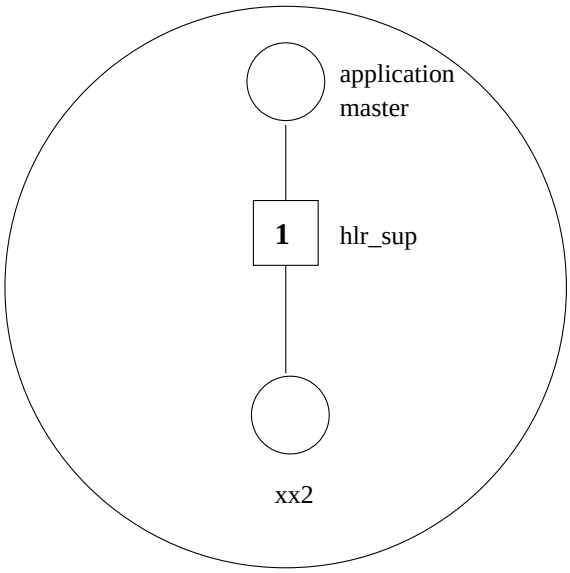


Figure 1.16: Illustration of HLR Application

Designing the Processes

Each application has an application callback module, with behaviour `application`. This module is called when the application is started, and when it has stopped. The `start` function in this module starts the topmost supervisor for the application.

All worker processes in the application should be written using the standard behaviours, such as `gen_server`, `gen_fsm` or `gen_event`. Alternatively, they should be special processes written with `sys` and `proc_lib`. There are two main reasons for this:

- we want to be able to change code on all processes
- we want a fault tolerant system.

Simple, temporary processes can be written as normal Erlang processes without using a standard behaviour. However, these processes must be started with `proc_lib:spawn_link` instead of the BIFs `spawn` or `spawn_link`. However, it will not be possible to change code in these processes.

> Note:
> Always use `spawn_link`, and never `spawn`. You might otherwise loose track of the process and produce a zombie process.

The next example illustrates the following scenario:

- we want a special alarm event handler installed during the lifetime of the `hlr` application
- this event handler is called `hlr_alarm_h` and writes each alarm to a special file
- when starting `hlr`, we want to install this event handler in the already existing alarm handler.

To implement this, we place the call `gen_event:add_handler(alarm_handler, hlr_alarm_h, FileName)` in the initialisation function of the application.

The application callback module for `hlr` will then look as follows:

```
-module(hlr).
-vsn(1).
-behaviour(application).

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

%%%----------------------------------------------------------------
%%% This module implements the application HLR.
%%%----------------------------------------------------------------
start(_, _) ->
    case hlr_sup:start() of
        {ok, Pid} ->
            gen_event:add_handler(alarm_handler, hlr_alarm_h, []),
            {ok, Pid, []};
        Error -> Error
    end.

stop(_State) ->
    gen_event:delete_handler(alarm_handler, hlr_alarm_h).
```

The supervisor will look as follows:

```
-module(hlr_sup).
-vsn(1).
-behaviour(supervisor).

%% External exports
-export([start/0]).

%% Internal exports
-export([init/1]).

%%%----------------------------------------------------------------
%%% This module implements a supervisor for the HLR application.
%%%----------------------------------------------------------------
start() ->
    supervisor:start_link({local, hlr_sup}, hlr_sup, []).

init([]) ->
    SupFlags = {one_for_one, 4, 3600},
    Vshlr = {xx2, {vshlr_2, start_link, []},
             permanent, 2000, worker, [vshlr_2]},
    {ok, {SupFlags, [Vshlr]}}.
```

Configuring the Application

In this section, the `hlr` alarm event handler is used as an example of how to configure an application. This handler is an instance of the `gen_event` behaviour. Its purpose is to write some alarms to a specified file. The event handler should be configured to write to the filename specified in the alarm output.

The file `hlr_alarms.cnf` contains a list of all alarms which should be logged. This file looks as follows:

```
hlr_almost_full.
hlr_inconsistent.
```

This file is stored in the private directory `priv` of the application. It is found by calling `code:priv_dir(hlr)`.

Each application has an associated environment where configuration parameters are defined. This environment is specified in the application specification, and is overridden by the system configuration file. The value of the parameter `hlr_alarm_file` is a string which specifies the file which logs all alarms. The value of the parameter is found with the call `application:get_env(hlr, hlr_alarm_file)`. The `hlr_alarm_h` looks as follows:

```
-module(hlr_alarm_h).
-vsn(1).
-behaviour(gen_event).

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

-record(state, {fd, alarms}).
%%----------------------------------------------------------------------
%% Callback functions from gen_event
%%----------------------------------------------------------------------
init(_) ->
    CnfFile = filename:join(code:priv_dir(hlr), "hlr_alarm.cnf"),
    Alarms = case file:consult(CnfFile) of
                 {ok, List} -> List;
                 _ -> []
             end,
    case application:get_env(hlr, hlr_alarm_file) of
        {ok, File} ->
            {ok, Fd} = file:open(File, write),
            {ok, #state{fd = Fd, alarms = Alarms}};
        undefined ->
            {error, {no_config, hlr_alarm_file}}
    end.

handle_event({set_alarm, Alarm}, State)->
    case is_hlr_alarm(Alarm, State) of
        true -> io:format(State#state.fd, "set alarm: ~p~n", [Alarm]);
        false -> ok
    end,
    {ok, State};
handle_event({clear_alarm, AlarmId}, State)->
    case is_hlr_alarm(Alarm, State) of
        true -> io:format(State#state.fd, "clear alarm: ~p~n", [AlarmId]);
        false -> ok
```

```
    end,
    {ok, State}.

handle_info(_, State) -> {ok, State}.

terminate(_, State) ->
    file:close(State#state.fd).

is_hlr_alarm({AlarmId, _}, #state{alarms = Alarms}) ->
    lists:member(AlarmId, Alarms).
```

Application Specification

An application specification is required in order to test the `hlr` application. This specification is placed in the file `hlr.app`, which looks as follows:

```
{application, hlr,
   [{description, "VSHLR"},
    {vsn, "1.0"},
    {modules, [{vshlr_2, 1}, {hlr_alarm_h, 1}, {hlr_sup, 1}, {hlr, 1}]},
    {registered, [hlr_sup, xx2]},
    {applications, [kernel, stdlib, sasl]},
    {env, [{hlr_alarm_file, "hlr.alarms"}]},
    {mod, {hlr, []}}]}.
```

This specification says that if no `hlr_alarm_file` is specified in the system configuration file, we use the file `hlr.alarms` in the current directory as a default.

Testing the Application

The next task is to test the application. In doing so, we also want to specify another `hlr` alarm file. We do this by writing a configuration file called `sys.config`:

```
[{hlr, [{hlr_alarm_file, "alarms.log"}]}].
```

The following interaction shows how to test the application. The command to start the system is followed by a command to start the application itself.

```
erl -pa . -config ./sys

5> application:start(hlr, temporary).

=PROGRESS REPORT==== 29-May-1996::14:04:05 ===
     Supervisor: {local,hlr_sup}
     Started:  [{pid,<0.54.0>},
                {name,xx2},
                {mfa,{vshlr_2,start_link,[]}},
                {restart_type,permanent},
                {shutdown,2000},
                {child_type,worker}]

ok
```

```
6> gen_event:which_handlers(alarm_handler).
[hlr_alarm_h,alarm_handler]
7> vshlr_2:i_am_at(martin, home).
ok
8> vshlr_2:find(martin).
{at,home}
9> application:stop(hlr).
ok
10> gen_event:which_handlers(alarm_handler).
[alarm_handler]
```

Distributed Applications

This section describes and illustrates how distributed applications can be used.

The example illustrated in this section is shown in [page 64].

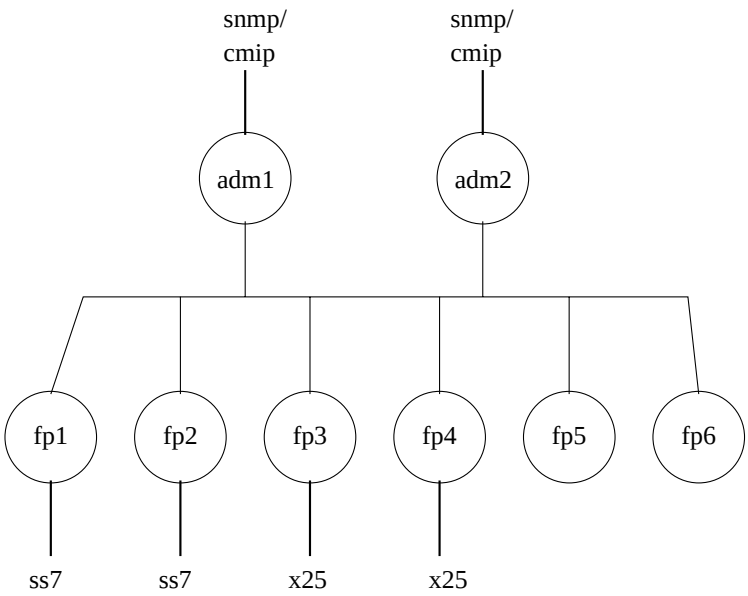


Figure 1.17: Example of Distributed Application

This system has the following components and characteristics:

- there are two administrative CPUs, `adm1` and `adm2`
- there are six functional CPUs, `fp1 - fp6` which are organized as shown in the illustration below
- the two administrative CPUs are used for redundancy and we want to use both of them for performance reasons
- there are five applications of different type:
    1. `snmp`. This is a management application, which interfaces an operator. There must be only one instance of this application in the system.

2. cmip. This is a management application, which interfaces an operator. There must be only one instance of this application in the system.
3. ch. This is a call handling application. It needs the applications ss7 and x25. We want as many instances of this application as possible, but only one per node.
4. ss7. This application interfaces ss7. We want as many instances of this application as possible, but only one per node.
5. x25. This application interfaces x25. There must only be one instance of this application in the system.

The administrative CPUs take care of the management applications snmp and cmip, and the functional CPUs the call handling application ch. This application uses the interfaces ss7 and x25, which are represented by corresponding applications. As shown in the figure, only fp1 and fp2 have an ss7 interface, and only fp3 and fp4 have an x25 interface.

This is summarized in the table below:

Application	Instances	Nodes
snmp	1	adm1, adm2
cmip	1	adm1, adm2
ch	N	fp1 - fp6
ss7	N	fp1, fp2
x25	1	fp3, fp4

Table 1.1: Node Distribution for Example Application

The following sections describe how to specify this parameter for the different applications in the example.

The SNMP and CMIP Applications

These applications can run on either adm1 or adm2. In normal operating mode, we want one application to run at each of these processors, snmp on adm1, and cmip on adm2. If one of the nodes goes down, the other node starts the application and both applications will run on one node. When the faulty node restarts, it takes over its application from the other node. This arrangement is specified as follows:

```
{snmp, [adm1, adm2]},
{cmip, [adm2, adm1]}
```

In the boot script, snmp and cmip is started on both nodes.

The CH Application

The ch application is a local application and is started in the boot script on each fp node. This call handling application is run on each fp node and the application controller can therefore not view this application as distributed.

The SS7 Application

This application also has several instances. For this reason, it cannot be distributed, but is local on nodes `fp1` and `fp2`.

The X25 Application

This is an application with one instance only and it can run on either `fp3` or `fp4`. Accordingly, it is a distributed application. In normal operating mode, we do not care on which one of these two processors the application runs, but if this node goes down, the other node must start the application. There is no need to move the application back to the original node if it restarts. This requirement is expressed as follows:

```
{x25, [{fp3, fp4}]}
```

1.9 Error Logging

The Erlang system has a process with the registered name `error_logger`. This process receives all error messages from the Erlang runtime system and error messages sent by the error reporting functions in the module `error_logger`.

This section describes the following topics:

- error types
- error message handling
- the standard error logger
- customized error report handlers.

Types of Errors

Errors are divided into the following three categories:

- predicted, recoverable errors
- predicted, unrecoverable errors
- unpredicted errors.

The system designer has to decide if a process can recover from predicted errors.

The last two categories of errors cause an abnormal termination of the process.

Error Message Handling

Errors can be reported by propagating an EXIT signal, by producing an error report, or both of these methods. The following table summarizes how the three categories of errors can be reported.

Category	EXIT reason	Error report
Predicted and recoverable	-	yes
Predicted and unrecoverable	yes	yes
Not predicted	yes	-

Table 1.2: Error Message Reporting

Recoverable errors are not reported by EXIT signals because the process does not terminate. Unpredicted errors are only reported by EXIT signals because the program does not take care of these errors.

Note:
The recovery from predicted errors is interpreted as an internal process error recovery. A process restart, issued by the layer above in the supervision tree, can also be used to recover from errors.

An unrecoverable, but predicted error can provide meaningful error descriptions, not only in an error report but also through an informative EXIT reason. It is recommended that the EXIT reason is composed as {Reason,{Mod,Fun,Args}} where:

- Reason is an informative error descriptor, for example an atom or {atom, Value}.
- {Mod,Fun,Args} is the Mod:Fun(Args) function where the error is detected.

The Standard Error Logger

The error_logger process receives and handles the following types of errors:

- all errors generated by the Erlang runtime system (the emulator)
- errors reported through the error_logger module interface functions error_msg/1, error_msg/2, info_msg/1, info_msg/2, format/2, error_report/1, and info_report/1.

When the error_logger process receives an error, it is sent to the the error_logger on the node which is the group leader process for the process which caused the error.

During system start-up, errors are kept in a buffer and they are also written unformatted to standard out, because error message handling is determined by the start-up process. The standard error_logger provides two possibilities:

- write to standard_out
- write to a specified file.

All buffered errors are written again when the intended handler is installed. Initially errors are written in the format {error_logger,Time,Arg1,Arg2}, where:

- Time is a tuple which contains date and time information
- Arg1 and Arg2 are the arguments given to a report function.

When a standard handler is installed, errors are written in the following format:

```
=ERROR REPORT=== Time ====
 Formatted error
```

Formatted error error messages, which are reported through the format or error_msg functions calls, are produced from Format and Args in the same way as in the function io:format/2. If the error was reported through the supplied error_report(Report) function, the Report argument is interpreted and written as follows:

- [{Tag,Info}], where Tag and Info is any term. Each {Tag,Info} tuple is written on a separate line:

```
Tag: Info
```

- Other is written as io_lib:format("~p~n",[Other]).

It is also possible to use the `info_report(Report)` function to format `Report` as `error_report/1` in the above example, but with the heading:

```
=INFO REPORT=== Time ====
```

Adding A Customized Report Handler

It is possible to add customized error report handlers to the `error_logger` process. This may be desirable in order to satisfy one of the following purposes:

- to perform additional processing of standard error messages
- to override the standard behaviour (the standard handlers must be deleted)
- to handle new types of error messages.

The following two functions are used to add and delete handlers:

- `add_report_handler` to add a handler
- `delete_report_handler` to delete a handler.

> Note:
> It is strongly recommended that the standard error reporting functions be used. Customized handlers should only be added if they are *really* needed.

The `error_logger` is implemented with the `gen_event` behaviour [page 37]. This means that an event callback module has to be implemented in order to add an associated error logger handler .

All handlers installed within the `error_logger` are notified about errors that are received by the `error_logger`. This notification is done by calling `handle_event/2` in each callback module. This means that several actions can be performed when specific events occur. For example, the standard `error_logger` behaviour can be accompanied by an SNMP trap which is triggered when a pre-determined level of error messages have been received.

New types of error message can be reported with the `error_report/2` and `info_report/2` functions. Errors which are reported this way are ignored by the standard error logger handlers and would be lost unless an associated handler has been installed.

The event generated by calling `error_logger:error_report(Type, Report)`, and which has to be handled by the added error report handler, is the term:

- `{error_report, Gleader, {Type, Pid, Report}}` where:
  - `Gleader` is the group leader process of the process which executes the function call.
  - `Type` is a term which identifies the type of the error report.
  - `Pid` is the process identity of the process that issued the error report. The `Pid` can be used to determine at which node the report was generated.
  - `Report` is a term which describes the error. This term must be recognized by the customized error report handler.

Substitute `info_report` in place of `error_report` when calling the `error_logger:info_report(Type, Report)` function.

> **Note:**
> No standard error_logger messages are described here.

The following example illustrates how an error report handler for `my_error` type of error messages can be implemented:

```
-module(my_error_logger_h).
-copyright('Copyright (c) 1991-97 Ericsson Telecom AB').
-vsn('$Revision: /main/release/2 $').
-behaviour(gen_event).

-export([start/0, stop/0, init/1, handle_event/2, handle_info/2,
         handle_call/2, terminate/2, report/1]).

start() -> error_logger:add_report_handler(my_error_logger_h).

stop() -> error_logger:delete_report_handler(my_error_logger_h).

report(My_error) ->
    error_logger:error_report(my_error, My_error).

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

handle_event({error_report, Gleader, {my_error, Pid, My_error}}, State) ->
    handle_my_error(Gleader, Pid, My_error),
    {ok, State};
handle_event(_, State) ->  % Ignore all other error messages.
    {ok, State}.

handle_info(_, State) ->
    {ok, State}.

handle_call(_, State) ->
    {error, bad_query}.

terminate(_, _) ->
    ok.

handle_my_error(Gleader, Pid, My_error) when node(Gleader) == node() ->
    %% do handle the error
    ok;
handle_my_error(_, _, _) -> % Ignore error if Gleader at another node.
    ok.
```

The purpose of the `error_logger` is to log or write errors. If some other type of event handler is needed it *must* be implemented as a customized process with another registered name. Refer to the section Events [page 37] for further information.

List of Figures

Chapter 1: Design Principles

List of Tables