Erlang – a platform for developing distributed software systems

Lars-Åke Fredlund
Problems of distributed systems

- Distributed programming is **hard**

- Challenges for concurrency:
  - process coordination and communication
Problems of distributed systems

- Distributed programming is **hard**

- Challenges for concurrency:
  - process coordination and communication

- And challenges for distributed software:
  - heterogeneous systems
  - security, reliability (lack of control)
  - performance
Distributed Programming Today

Todays contrasts:

■ Vision – easy programming of distributed systems

■ The nightmare/reality – Web services, XML, Apache, SOAP, WSDL, …

■ Why are Web services a nightmare?
  Too many standards, too many tools, too many layers, too complex!

■ Erlang is an Industrially proven solution for developing and maintaining demanding distributed applications

■ Good qualities of Erlang as a distributed systems platform:
  Complexity encapsulated in a programming language, good performance, efficient data formats, debuggable, not complex
Erlang as a component platform: a summary

- Processes are the components
Processes are the components

Components (processes) communicate by binary asynchronous message passing
Erlang as a component platform: a summary

- **Processes** are the **components**
- Components (processes) communicate by **binary asynchronous message passing**
- Component communication does not depend on whether components are located in the same node, or physically remote (**distribution is seamless**)
Erlang as a component platform: a summary

- **Processes** are the components

- Components (processes) communicate by **binary asynchronous message passing**

- Component communication does not depend on whether components are located in the same node, or physically remote (**distribution is seamless**)

- Component programming is facilitated by using **design patterns** (client/server patterns, patterns for fault tolerant systems, etc) and **larger components** (web server, database)
Processes are the components

Components (processes) communicate by binary asynchronous message passing

Component communication does not depend on whether components are located in the same node, or physically remote (distribution is seamless)

Component programming is facilitated by using design patterns (client/server patterns, patterns for fault tolerant systems, etc) and larger components (web server, database)

Component maintenance, and fault tolerance is facilitated by language features and design patterns
Erlang as a component platform: a summary

- **Processes** are the **components**

- Components (processes) communicate by **binary asynchronous message passing**

- Component communication does not depend on whether components are located in the same node, or physically remote (distribution is seamless)

- Component programming is facilitated by using **design patterns** (client/server patterns, patterns for fault tolerant systems, etc) and **larger components** (web server, database)

- Component **maintenance**, and **fault tolerance** is facilitated by language features and design patterns

- But the devil is in the details: let’s see them!
Erlang/OTP

- Basis: a general purpose functional programming language
- Automatic Garbage Collection
- With lightweight processes
  (in terms of speed and memory requirements)
- Typical software can make use of many thousands of processes; **smp** supported on standard platforms
- Implemented using virtual machine technology and compilation to native code (Intel x86, Sparc, Power PC)
  Available on many OS:es (Windows, Linux, Solaris, ...)
- Supported by extensive libraries:
  **OTP** – open telecom platform – provides tools such as components, distributed database, web server, etc
Erlang/OTP History

- Erlang language born in 1983
- Used inside and outside Ericsson for telecommunication applications, for soft real-time systems, …
- Industrial users: Ericsson, Swedish Telecom, T-Mobile (UK), and many smaller start-up companies (LambdaStream in A Coruña)
- Application example: High-speed ATM switch developed in Erlang (2 million lines of Erlang code), C code (350 000 lines of code), and 5 000 lines of Java code
- Other examples: parts of Facebook chat written in Erlang (70 million users), CouchDB (integrated in Ubuntu 9.10), users at Amazon, Yahoo, …
- Open-source; install from http://www.erlang.org/
Erlang is becoming popular

C and C++ job offers over the last 5 years:

Erlang job offers the last 5 years:
Concurrency and communication model from Erlang are also influencing other programming languages and libraries like Scala, Node.js, Clojure, …

So let's see the main features…
Erlang basis

A simple functional programming language:

- Simple data constructors:
  - integers (2), floats (2.3), atoms (hola), tuples ({2, hola})
  - and lists ([2, hola], [2 | X]), records
    (#process{label=hola}), bit strings (<<1:1, 0:1>>)

- Call-by-value

- Variables can be assigned once only (Prolog heritage)

- No static type system!
  - That is, expect runtime errors and exceptions

- Similar to a scripting language (python, perl) – why popular?
Example:

\[
\text{fac}(N) \rightarrow \\
\quad \text{if} \\
\quad \quad N == 0 \rightarrow 1; \\
\quad \quad \text{true} \rightarrow N \times \text{fac}(N-1) \\
\quad \text{end.}
\]

Variables begin with a capital (N)
Atoms (symbols) begin with a lowercase letter (fac, true)
Example:

\[
\text{fac}(N) \rightarrow \\
\text{if} \\
N == 0 \rightarrow 1; \\
\text{true} \rightarrow N \times \text{fac}(N-1) \\
\text{end}.
\]

Variables begin with a capital (N)
Atoms (symbols) begin with a lowercase letter (fac, true)

But this also compiles without warning:

\[
\text{fac}(N) \rightarrow \\
\text{if} \\
N == 0 \rightarrow 1; \\
\text{true} \rightarrow "upm" \times \text{fac}(N-1) \\
\text{end}.
\]
Example:

```erlang
fac(N) ->
    if
        N == 0 -> 1;
        true -> N*fac(N-1)
    end.
```

Variables begin with a capital (N)
Atoms (symbols) begin with a lowercase letter (fac,true)

But this also compiles without warning:

```erlang
fac(N) ->
    if
        N == 0 -> 1;
        true -> "upm"*fac(N-1)
    end.
```

And this call is permitted (what happens?): `fac(0.5)`
Concurrent and Communication model inspired by the *Actor model* (and earlier Ericsson software/hardware products)

- Processes execute Erlang functions
- No implicit sharing of data (shared variables) between processes
- Two interprocess communication mechanisms exists:
  - Processes can send asynchronous messages to each other (*message passing*)
  - Processes get notified when a related process dies (*failure detectors*)
Erlang Processes

- Processes execute Erlang functions \((f(Arg_1, \ldots, Arg_n))\)
- A process has a unique name, a **process identifier** \((Pid)\)
- Messages sent to a process is stored in a **mailbox** \((M_2, M_1)\)
Sending a message to a process:

\[
\text{Pid}!\{\text{request, self()}, \text{a}\}
\]

Retrieving messages from the process mailbox (queue):

\[
\text{receive} \\
\begin{align*}
\{\text{request, RequestPid, Resource}\} & \rightarrow \\
\text{lock(Resource), RequestPid!ok} \\
\end{align*}
\]

Creating a new process:

\[
\text{spawn}\left(\text{fun} () \rightarrow \text{locker}!\{\text{request}, \text{B}\}\right)\]

A name server assigns symbolic names to processes:

\[
\text{locker}!\{\text{request}, \text{a}\}\]
Communication Primitives, receiving

Retrieving a message from the process mailbox:

```plaintext
receive
  \text{pat}_1 \text{ when } g_1 \rightarrow \text{expr}_1 ;
  \ldots ;
  \text{pat}_n \text{ when } g_n \rightarrow \text{expr}_n
  \text{after } \text{time} \rightarrow \text{expr'}
end
```

- \text{pat}_1 is matched against the oldest message, and checked against the guard \( g_1 \). If a match, it is removed from the mailbox and \( \text{expr}_1 \) is executed.

- If there is no match, pattern \( \text{pat}_2 \) is tried, and so on...

- If no pattern matches the first message, it is kept in the mailbox and the second oldest message is checked, etc.

- \text{after} provides a timeout if no message matches any pattern.
Receive Examples

Given a receive statement:

```
receive
  {inc,X} -> X+1;
  Other -> error
end
```

and the queue is \( a \cdot \{inc, 5\} \) what happens?
Receive Examples

- Given a receive statement:
  ```
  receive
      {inc,X} -> X+1;
      Other -> error
  end
  ```

  and the queue is $a \cdot \{inc, 5\}$ what happens?

- Suppose the queue is $a \cdot \{inc, 5\} \cdot b$ what happens?
Receive Examples

- Given a receive statement:
  
  ```plaintext
  receive
    {inc,X} -> X+1;
    Other -> error
  end
  
  and the queue is $a \cdot \{inc, 5\}$ what happens?
  
- Suppose the queue is $a \cdot \{inc, 5\} \cdot b$ what happens?
  
- Suppose the receive statement is
  
  ```plaintext
  receive
    {inc,X} -> X+1
  end
  
  and the queue is $a \cdot \{inc, 5\} \cdot b$ what happens?
Receive Examples

- Given a receive statement:
  ```
  receive
    {inc,X} -> X+1;
    Other  -> error
  end
  ```

  and the queue is $a \cdot \{inc, 5\}$ what happens?

- Suppose the queue is $a \cdot \{inc, 5\} \cdot b$ what happens?

- Suppose the receive statement is
  ```
  receive
    {inc,X} -> X+1
  end
  ```

  and the queue is $a \cdot \{inc, 5\} \cdot b$ what happens?

- And if the queue is $a \cdot b$?
Messages sent from any process P to any process Q is delivered in order (or P or Q crashes)
But the following situation is possible:

- Process P sends a message M1 to process Q
- and P then a message M2 to process R
- R forwards the message M2 to Q
- **Process Q may receive M2 from R before M1 from Q**

Mimics TCP/IP communication guarantees
facserver() ->
  receive
    {request, N, Pid}
    when is_integer(N), N>0, pid(Pid) ->
      spawn(fun () -> Pid!(fac:fac(N)) end),
      facserver()
  end.
A Simple Concurrent Program

facserver() ->
    receive
      {request, N, Pid}
      when is_integer(N), N>0, pid(Pid) ->
        spawn(fun () -> Pid!(fac:fac(N)) end),
        facserver()
    end.

1> spawn(server, facserver, []).
<0.33.0>
A Simple Concurrent Program

```erlang
facserver() ->
    receive
        {request, N, Pid} when is_integer(N), N>0, pid(Pid) ->
            spawn(fun () -> Pid!(fac:fac(N)) end),
            facserver()
    end.
```

1> spawn(server, facserver, []).
<0.33.0>

2> X = spawn(server, facserver, []).
<0.35.0>
A Simple Concurrent Program

```
facserver() ->
    receive
        {request, N, Pid} when is_integer(N), N>0, pid(Pid) ->
        spawn(fun () -> Pid!(fac:fac(N)) end),
        facserver()
    end.
```

```
1> spawn(server,facserver,[]).
<0.33.0>
2> X = spawn(server,facserver,[]).
<0.35.0>
3> X!{request,2,self()}.  
   {request,2,<0.31.0>}
```
A Simple Concurrent Program

```erlang
facserver() ->
receive
    {request, N, Pid}
    when is_integer(N), N>0, pid(Pid) ->
        spawn(fun () -> Pid!(fac:fac(N)) end), facserver()
end.
```

1> `spawn(server,facserver,[]).`
<0.33.0>

2> `X = spawn(server,facserver,[]).`
<0.35.0>

3> `X!{request,2,self()}.`
{request,2,<0.31.0>}

4> `X!{request,4,self()}, receive Y -> Y end.`
2
Erlang and Errors

- Unavoidably errors happen in distributed systems
Unavoidably errors happen in distributed systems

- hardware (computers) fail
Erlang and Errors

- Unavoidably errors happen in distributed systems
  - hardware (computers) fail
  - network links fail
Unavoidably errors happen in distributed systems

- hardware (computers) fail
- network links fail
- local resources (memory) runs out
Erlang and Errors

- Unavoidably errors happen in distributed systems
  - hardware (computers) fail
  - network links fail
  - local resources (memory) runs out

- Errors happen, good fault-tolerant systems cope with them
Erlang and Errors

- Unavoidably errors happen in distributed systems
  - hardware (computers) fail
  - network links fail
  - local resources (memory) runs out

- Errors happen, good fault-tolerant systems cope with them

- Many Erlang products have high availability goals: 24/7, 99.9999999% of the time for the Ericsson AXD 301 switch (31 ms downtime per year!)
Erlang and Errors

- Unavoidably errors happen in distributed systems
  - hardware (computers) fail
  - network links fail
  - local resources (memory) runs out

- Errors happen, good fault-tolerant systems cope with them

- Many Erlang products have high availability goals: 24/7, 99.9999999% of the time for the Ericsson AXD 301 switch (31 ms downtime per year!)

- The Erlang philosophy is to do error detection and recovery, but not everywhere in the code, only in certain places
Erlang and Errors

■ Unavoidably errors happen in distributed systems
  ◆ hardware (computers) fail
  ◆ network links fail
  ◆ local resources (memory) runs out

■ Errors happen, good fault-tolerant systems cope with them

■ Many Erlang products have high availability goals: 24/7, 99.9999999% of the time for the Ericsson AXD 301 switch (31 ms downtime per year!)

■ The Erlang philosophy is to do error detection and recovery, but not everywhere in the code, only in certain places

■ Higher-level Erlang components offer convenient handling of errors
Error handling example:

\[
g(Y) ->
\quad X = f(Y),
\quad \text{case } X \text{ of}
\quad \quad \{\text{ok, Result}\} -> \text{Result};
\quad \quad \text{reallyBadError} -> 0 \% \text{ May crash because of ...}
\quad \text{end.}
\]
Error handling example:

```erlang
\[
g(Y) ->
    X = f(Y),
    case X of
        {ok, Result} -> Result;
        reallyBadError -> 0 \% May crash because of ...
    end.
\]
```

instead one usually writes

```erlang
\[
g(Y) ->
    {ok, Result} = f(Y), Result.
\]
```
Error handling example:

```erlang
\[ g(Y) ->
    X = f(Y),
    case X of
        {ok, Result} -> Result;
        reallyBadError -> 0 \% May crash because of ...
    end.
\]
```

instead one usually writes

```erlang
\[ g(Y) ->
    {ok, Result} = f(Y), Result.
\]
```

The local process will crash; another process is responsible from recovering (restaring the crashed process)
Error handling example:

g(Y) ->
    X = f(Y),
    case X of
        {ok, Result} -> Result;
        reallyBadError -> 0  %% May crash because of ...
    end.

Instead one usually writes

g(Y) ->
    {ok, Result} = f(Y), Result.

The local process will crash; another process is responsible from recovering (restaring the crashed process)

Error detection and recovery is localised to special processes, to special parts of the code (aspect oriented programming)
Exceptions are generated at runtime due to:

- type mismatches ($10 \ast "\text{upm}"$)
- failed pattern matches, processes crashing, …

Exceptions caused by an expression $e$ may be recovered inside a process using the construct `try e catch m end`

Example:
```
try
g(Y)
catch
  Error \rightarrow 0
end
```
Within a set of processes, via bidirectional process links set up using the `link(pid)` function call

Example:
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the `link(pid)` function call

- Example:

  Initially we have a system of 3 independent processes:

  ![Diagram](image)
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the `link(pid)` function call

- Example:

  Result of executing `link(P1)` in `P2`:

```plaintext
P1 → P2

P2 → P1

P3
```

P1 → P2 → P3
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the \texttt{link(pid)} function call

- Example:

  Result of executing \texttt{link(P1)} and \texttt{link(P3)} in \texttt{P2}:
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the `link(pid)` function call

- Example:

  Result of executing `link(P1)` and `link(P3)` in `P2`:

  ![Diagram](image)

- If `P2` dies abnormally then `P1` and `P3` can choose to die
- If `P1` dies abnormally then `P2` can choose to die as well
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the \texttt{link(pid)} function call

- Example:

  Result of executing \texttt{link(P1)} and \texttt{link(P3)} in \texttt{P2}:

  \begin{center}
  \begin{tikzpicture}
  \node [fill=blue!20, circle] (P2) at (0,0) {$P_2$};
  \node [fill=blue!20, circle] (P1) at (-1,-2) {$P_1$};
  \node [fill=blue!20, circle] (P3) at (1,-2) {$P_3$};
  \draw[->] (P2) -- (P1);
  \draw[->] (P2) -- (P3);
  \end{tikzpicture}
  \end{center}

  - If \texttt{P2} dies abnormally then \texttt{P1} and \texttt{P3} can \textit{choose} to die
  - If \texttt{P1} dies abnormally then \texttt{P2} can \textit{choose} to die as well
  - Alternatively when \texttt{P2} dies both \texttt{P1} and \texttt{P3} receives a message concerning the termination
What is Erlang suitable for?

- Generally intended for long-running programs
- Processes with state, that perform concurrent (and maybe distributed) activities
- Typical is to have a continuously running system (24/7)
- Programs need to be fault-tolerant (because hardware and software invariably fail)
- So hardware is typically replicated as well – and thus we have a need for distributed programming (addressing physically isolated processors)
Distributed Erlang

- Processes run on nodes (computers) in a network

- Distribution is (mostly) transparent
  - No syntactic difference between inter-node or intra-node process communication
  - Communication link failure or node failures are interpreted as process failures (detected using linking)
Distributed Erlang

- Processes run on nodes (computers) in a network

- Distribution is (mostly) transparent
  - No syntactic difference between inter-node or intra-node process communication
  - Communication link failure or node failures are interpreted as process failures (detected using linking)
  - Compare with Java: no references to objects which are difficult to communicate in messages (copy?)
  - The only references are process identifiers which have the same meaning at both sending and receiving process
Erlang Programming Styles

- Using only the basic communication primitives (send/receive) makes for messy code – everybody invents their own style and repeats lots of code for every program.

- We need at least a standard way to:
  - ask processes about their status
  - a standard way to handle process start, termination and restarts
  - to handle code upgrading
  - and maybe more structured communication patterns: who communicates with whom, in what role?…
We need to structure the code according to some more design principles, to obtain more ”regular” code.

For Erlang one generally uses the components and the framework of the **OTP library – Open Telecom Platform** – as an infrastructure.

Today we are going to illustrate a number of these design principles, and how they are used in practise.
OTP – an Erlang library of components

- A library of components for typical programming patterns (e.g., client–server, managing processes, . . .)

- In contrast to many component frameworks OTP is not concerned with how to link components together but with:
  - operation and management of components
  - fault-handling for components
OTP – an Erlang library of components

- A library of components for typical programming patterns (e.g., client–server, managing processes, …)

- In contrast to many component frameworks OTP is not concerned with how to *link components together* but with:
  - operation and management of components
  - fault-handling for components

- OTP components uses similar behaviour wrt management concerns such as
  - Starting a component
  - Terminating a component
  - Dynamic code update (change code at runtime)
  - Inspecting components
  - Handling errors
Component/Behaviour Style

- Declarative specifications are preferred

- **Callback style** – Component descriptions are composed of two parts:
  - A generic part containing the generic component code
  - A concrete one where the default behaviour is specialised to the concrete application by supplying **function definitions**

- As a result: a weak object-orientation style (very weak type checking of component specialisation)

- Except it is based on processes, a pretty powerful concept
OTP components

Example OTP components:

- **Application**
  - provides bigger building blocks like a database (Mnesia)

- **Supervisor**
  - used to start and bring down a set of processes, and to manage processes when errors occur

- **Generic Server**
  - provides a client–server communication facility

- **Event Handling**
  - for reporting system events to interested processes

- **Finite State Machine**
  - provides a component facilitating the programming of finite state machines in Erlang
Event Handling: Processes

- An implementation of a publish-and-subscribe behaviour
- An *Event manager* controls the publishing of events
- *Event handlers* register interest to receive events from a particular event manager by sending a message to the event manager
- Some process generates an event, which is sent to all the interested event handlers (*Event generator*)
(1): The event handlers registers themselves:
(2): Some process generates an event:

Event Manager

Event Generator

Event Generator

Event Handler

Event Handler

Event Handler
(3): The event is handled by the event handlers:
Events can be delivered synchronously or asynchronously. The synchronous case means returning a reply to the event generator:
Events can be delivered synchronously or asynchronously. The *synchronous* case means returning a reply to the event generator:

(4): All event handlers return their status to the event handler when they have finished:
Events can be delivered synchronously or asynchronously. The *synchronous* case means returning a reply to the event generator:

(5): And the event handler tells the event generator when it has finished its execution.
Event Handling, behaviour overview

- Works distributedly, like all other components (behaviours)
- Includes managerial aspects:
  - Error handling: what happens if an event handler crashes?
  - Permits changing event handlers
  - Permits shutting down event handlers
  - Includes code upgrade facility
Applications are often structured as *supervision trees*, consisting of *supervisors* and *workers*

A supervisor starts child processes, monitors them, handles termination and stops them on request

The actions of the supervisor are described in a declarative fashion (as a text description)

A child process may itself be a supervisor
Supervision Dynamics

When a child process C1 dies (due to an error condition), its supervisor S3 is notified and can elect to:

- do nothing
- itself die (in turn notifying its supervisor S)
- restart the child process (and maybe its siblings)
- kill all the sibling processes (C2, C3) of the dead process
When a child process C1 dies (due to an error condition), its supervisor S3 is notified and can elect to:

- do nothing
- itself die (in turn notifying its supervisor S)
- restart the child process (and maybe its siblings)
- kill all the sibling processes (C2,C3) of the dead process

One can control the frequency of restarts, and the maximum number of restarts to attempt – it is no good having a process continuing to restart and crash.
Supervision Examples

- A file streaming application:

If the sender crashes, its supervisor restarts it (and vice versa for the receiver)
Supervision Examples

- A file streaming application:

  ![Diagram](image)

  If the sender crashes, its supervisor restarts it (and vice versa for the receiver)

- A file transfer application (with acknowledgment handling):

  ![Diagram](image)

  If the sender crashes, both it and the receiver is restarted
■ **gen_server** is *the* most used component in Erlang systems

■ Provides a standard way to implement a server process, and interface code for clients to access the server

■ The client–server model has a central server, and an arbitrary number of clients:
The Generic Server Component

- Clients make *requests* to the server, who optionally *replies*.
- A server has a state, which is preserved between requests.
- A generic server is implemented by providing a callback module specifying the concrete actions of the server (server state handling, and response to messages).
(1): A client sends a request:
(2): The server does some internal processing to answer, resulting in a new server state **State1**:
(3a): And eventually sends the reply to the client:
(3b): Or it doesn’t send a reply, but may do so in the future, and in the meanwhile accepts a new request:
Client interface:

- `Res = gen_server:call(ServerName, Message)`
  A call to `ServerName` (a pid) with a return value

- `gen_server:cast(ServerName, Message)`
  When no return value is expected
Generic Server: server interface

Server interface:

- **init(Args)** – at startup, returns the initial state of the server

- **handle_call(Message, ClientId, ServerState)**
  called when a `gen_server:call` is made. `ServerState` is the current state of the server.
  Should return a new server state

- **handle_cast(Message, ServerState)**
  called when a `gen_server:cast` is made.
  Should return a new server state
Generic Server: server returns

Server return values:

- \{\text{reply, Value, NewState}\}
  server replies with Value, and new server state is NewState

- \{\text{noreply, NewState}\}
  server send no reply (yet), but may do so in the future, the new server state is NewState

- \{\text{stop, Value}\}
  server stops but first returns Value to the current request
A simple generic server example

- We want to implement a simple server `locker` that grants access to a resource for only a single client at a time.

- Clients request access to the server using a message `request`.

- Once a client has finished with the resource it is released by sending the message `release`.

- A client function that requests the resource, applies a function $F$ on the resource, and then releases:

  ```erlang
  client(F) ->
  {ok, Resource} = gen_server:call(locker, request),
  %% We have resource, call $F$
  NewResource = apply(F, [Resource]),
  gen_server:call(locker, {release, NewResource}).
  ```
The state of the server is a tuple of two components:

- the current value of the resource, and
- a list where the first element is the client currently accessing the resource, and the rest of the list is the queue of clients wanting to access it.

The initial state is \( \{\text{Res, } []\} \) – no clients accessing.

An example state: \( \{\text{Res, [Pid1, Pid2, Pid3]}\} \) – Pid1 is accessing the resource, Pid2, Pid3 are awaiting their turn.
Server side: callback module example

init(Res) -> {ok, {Res, []}}.
%% No clients queued

handle_call(request, Client, {Res, Queue}) ->
  if
    Queue==[] -> {reply, {ok,Res}, {Res, [Client]}};
    Queue=/=[ ] -> {noreply, {Res, Queue++[Client]}}
  end;

handle_call>({release,Res}, Client, {_ , Queue}) ->
  case Queue of
    [Client] ->
      {reply, done, {Res,[]}};
    [Client,FirstWaiter|RestQueue] ->
      gen_server:reply(FirstWaiter, {ok,Res}),
      {reply, done, {Res, [FirstWaiter|RestQueue]}}
  end.
example: handling errors

- But what happens if the client crashes while it has access to the resource
example: handling errors

- But what happens if the client crashes while it has access to the resource
- ... – well the server will stay locked for ever
example: handling errors

- But what happens if the client crashes while it has access to the resource
- …– well the server will stay locked for ever
- We had better handle this case; the callback function handle_info will be called whenever a linked process terminates
Handling Errors in the example

Modifying `handle_call` to link to the client requesting access:

```erlang
handle_call(request, Client, {Res, Queue}) ->
    link(Client),
    if
        Queue==[] -> {reply, {ok, Res}, {Res, [Client]}},
        Queue=/=[_] -> {noreply, {Res, Queue++[Client]}}
    end;
```
Handling Errors in the example

Adding the function which handles errors:

```erlang
handle_info({'EXIT', Client, _}, {Res, Queue}) ->
    case Queue of
        [Client, FirstWaiter | RestQueue] ->
            gen_server:reply(FirstWaiter, {ok, Res}),
            {noreply, {Res, [FirstWaiter | RestQueue]}};
        _ ->
            {noreply, {Res, remove(Client, Queue)}}
    end.
```
Error Handling in the Generic Server component

Note some nice properties of error handling in generic servers:

- only handling errors in one (1) place in the code
- only handling errors at very controlled points in time (when not processing a request)
- We control error handling – we do not letting error handling control us!
- Such separation of concerns (between error handling and normal processing) is the real key to the power of the OTP components!
Generic Server Actions

- Not shown:
  - handling timeouts

- Generic behaviours handled mostly automatically by the component:
  - How to trace and log the actions of the server
  - How to terminate and restart a server
Since components are alive for a long time, it may be necessary to update the code of a component (its implementation) during its lifetime.

The generic server behaviour, like other Erlang behaviours, offers a standard method to do this:

- Upgrades are handled through the `code_change(Info1, OldState, Info2)` callback function which is called when a code change has taken place.
- `OldState` is the state of the server running the old version of the code.
- The callback should return a tuple `{ok, NewState}`.
Suppose that we want to add a field `NumOfRequest` to the server state for counting the number of requests made to the resource.

Recall that the state is `{Res, WaitingClients}`.

To do a code upgrade we provide in the new server implementation the function:

```erb
code_change(_, {Res, WaitingClients}, _) ->
  {ok, {Res, WaitingClients,
       length(WaitingClients)}}.
```
The generic server component processes messages in strict sequential (oldest first) order.

**Good:** makes for good performance (no searching of mailbox), bounded queues (no messages left in queue).

**Bad:** can make for complex processing logic (e.g., how to cleanly implement a one-bit buffer with two messages: `push` and `pop`).

Normally leads to a more complex server state (having queues inside the server state).

Other more stateful components are possible, accepting different messages at different times.

But how is low-level performance impacted, and how are unexpected messages handled (growing queues)?

A central problem in the design of Erlang processes!
Erlang/OTP Tools

- **Mnesia database** – relational/object data model, soft-real time properties, transactions, language integration, persistence, monitoring …

- **Yaws web server** – for serving dynamic content produced by Erlang code (good performance, elegant – everything written in Erlang; no need for Perl)

- **Interfaces** to other applications and systems: SQL databases, libraries for communicating with Java, XML parsers…
  
  ◆ languages: port concept
  
  ◆ databases

- And SASL (release upgrade, alarm handling), SNMP, …
Validating Erlang Programs

- **Dialyzer** – type checking by static analysis (necessary because of dynamic run-time typing)

- As usual, testing: **QuickCheck** (http://www.quiviq.com) - a testing tool both for the sequential and the concurrent part of Erlang

- Trace log inspection (ad-hoc)

- Model checking – my tool **McErlang** (http://babel.ls.fi.upm.es/~fred/McErlang)