

# Message-Passing Concurrency in Erlang

**Lecture 7** of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG

---



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Today's menu

- Actors and message passing
- Sending and receiving messages
- Stateful processes
- Clients and servers
- Generic servers
- Location transparency & distribution

# What is Erlang?

**Erlang** combines a **functional language** with **message-passing** features:

- The functional part is **sequential**, and is used to define the behavior of **processes**
- The message-passing part is highly **concurrent**: it implements the **actor model**, where actors are Erlang processes

This lecture covers the **message-passing/concurrent** part of Erlang

# ACTORS AND MESSAGE PASSING

# Erlang's Principles

**Concurrency** is **fundamental** in Erlang, and it follows models that are quite different from those offered by most imperative languages

In **Erlang** (from Armstrong's PhD thesis):

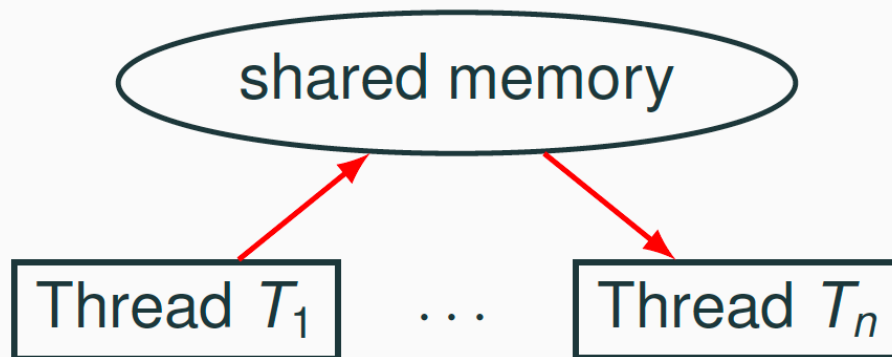
- Processes are strongly isolated
- Process creation and destruction is a lightweight operation
- Message passing is the only way for processes to interact
- Processes have unique names
- If you know the name of a process, you can send it a message
- Processes share no resources
- Error handling is non-local
- Processes do what they are supposed to do or fail

Compare these principles to programming using Java threads!

# Shared Memory vs. Message Passing

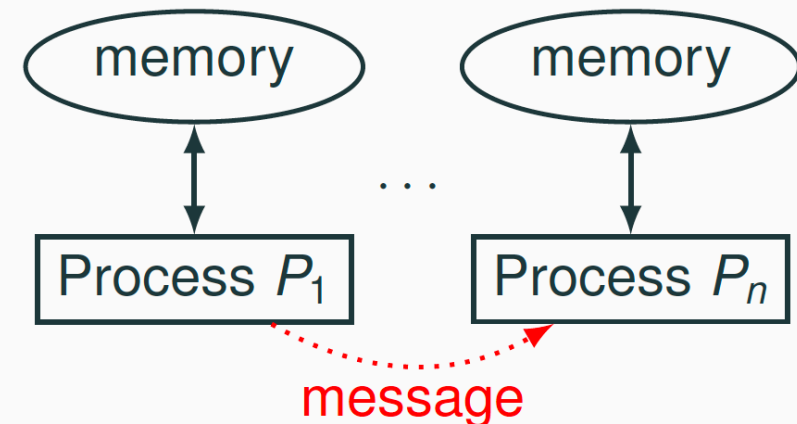
## Shared memory:

- synchronize by **writing to** and **reading from shared** memory
- natural choice in shared memory systems such as threads



## Message passing:

- synchronize by **exchanging messages**
- natural choice in distributed memory systems such as processes



# The Actor Model

Erlang's message-passing concurrency mechanisms implement the **actor model**:

- **Actors** are abstractions of processes
- **No shared state** between actors
- Actors **communicate** by **exchanging messages** – **asynchronous** message passing

A metaphorical **actor** is an “active agent which plays a role on cue according to a script” (Garner & Lukose, 1989)

# Actor and Messages

Each actor is identified by an **address**

An **actor** can:

- **send** (finitely many) **messages** to other actors via their addresses
- **change** its **behavior** – what it computes, how it reacts to messages
- **create** (finitely many) **new actors**

A **message** includes:

- a **recipient** – identified by its address
- **content** – arbitrary information



# The Actor Model in Erlang

The entities in the actor model correspond to **features of Erlang** (possibly with some terminological change)

ACTOR MODEL	Erlang	LANGUAGE
actor	sequential process	
address	PID (process identifier)	pid type
message	an Erlang term	{From, Content}
behavior	(defined by) functions	
create actor	spawning	<code>spawn</code>
dispose actor	termination	
send message	send expression	To ! Message
receive message	receive expression	<code>receive...end</code>

# SENDING AND RECEIVING MESSAGES

# A Process's Life

A **process**:

- is **created** by calling **spawn**
- is identified by a **pid** (process identifier)
- **executes** a function (passed as argument to **spawn**)
- when the function terminates, the process **ends**

# The spawn function

Function `spawn`(M, F, Args) creates a **new process**:

- the process runs function F in module M with arguments Args
- evaluating `spawn` returns the pid of the created process

Within a process's code, function `self` () returns the process's **pid**

Within a module's code, macro `?MODULE` gives the **module's name**

Calling `spawn (fun () -> f (a1, ..., an) end)` is equivalent to  
`spawn (?MODULE, f, [a1, ..., an])` but does not require exporting `f`

# Processes: Examples

A process code:

```
-module(procs).
```

```
print_sum(X,Y) ->
```

```
    io:format("~p~n", [X+Y]).
```

```
compute_sum(X,Y) -> X + Y.
```

```
3> spawn(fun () -> true end).
```

```
<0.82.0> % pid of spawned process
```

```
4> self().
```

```
<0.47.0> % pid of process running shell
```

Creating processes in the shell:

```
3> spawn(procs, print_sum, [3, 4]).
```

```
7          % printed sum
```

```
<0.78.0> % pid of spawned process
```

```
2> spawn(procs, compute_sum, [1, 7]).
```

```
<0.80.0> % pid of spawned process
```

```
          % result not visible!
```

# Sending Messages

A **message** is any **term** in Erlang

Typically, a message is the result of **evaluating** an expression

The expression

`Pid ! Message`

"Bang" operator



**sends** the evaluation  $T$  of `Message` to the process with pid `Pid`; and returns  $T$  as result

**Bang** is right-associative

To send a message to multiple recipients, we can combine multiple bangs:

```
Pidn1 ! Pidn2 ! ... ! Pidn ! Message
```

# Mailboxes

Every process is equipped with a **mailbox**, which behaves like a FIFO **queue** and is filled with the **messages** sent to the process in the order they arrive.

Mailboxes make **message-passing asynchronous**: the sender does not wait for the recipient to receive the message; messages queue in the mailbox until they are processed

To check the content of process Pid's mailbox, use functions:

- `process_info(Pid, message_queue_len)`: how many elements are in the mailbox
- `process_info(Pid, messages)`: list of messages in the mailbox (oldest to newest)
- `flush()`: empty the current process's mailbox

```
1> self() ! self() ! hello.    % send 'hello' twice to self
2> self() ! world.            % send 'world' to self
3> erlang:process_info(self(), messages)
{messages, [hello, hello, world]}    % queue in mailbox
```

# Receiving messages

To **receive messages**, use the **receive** expression:

```
receive  
  P1 when C1 -> E1;  
  ⋮  
  Pn when Cn -> En  
end
```

Evaluating the **receive** expression selects the **oldest** term  $\mathbb{T}$  in the receiving process's mailbox that matches a pattern  $P_k$  and satisfies condition  $C_k$

If a term  $\mathbb{T}$  that matches exists, the **receive** expression evaluates to  $E_k\langle P_k \triangleq \mathbb{T} \rangle$ ; otherwise, evaluation **blocks** until a suitable message arrives



# The receiving algorithm

How evaluating **receive** works, in pseudo-code:

```
Term receive (Queue<Term> mailbox, List<Clause> receive) {
  while (true) {
    await (!mailbox.isEmpty()); // block if no messages
    for (Term message: mailbox) // oldest to newest
      for (Clause clause: receive) // in textual order
        if (message.matches (clause.pattern))
          // apply bindings of pattern match
          // to evaluate clause expression
          return clause.expression <clause.pattern $\hat{=}$ message> ;
  }
}
```

# Receiving messages: examples

A simple echo function, which prints any message it receives:

```
echo () ->
  receive Msg -> io:format("Received: ~p~n", [Msg]) end.
```

Sending messages to echo in the shell:

```
1> Echo=spawn(echo, echo, []).
% now Echo is bound to echo's pid
2> Echo ! hello. % send 'hello' to Echo
Received: hello % printed by Echo
```

To make the receiving process permanent, it calls itself after receiving:

```
repeat_echo () ->
  receive Msg -> io:format("Received: ~p~n", [Msg]) end,
  repeat_echo(). % after receiving, go back to listening
```

← tail recursive, thus no memory consumption problem!

# Message delivery order

Erlang's runtime only provides weak guarantees of **message delivery order**:

- If a process S sends some messages to **another process** R, then R will receive the messages in the **same order** S sent them
- If a process S sends some messages to **two (or more)** other processes R and Q, there is **no guarantee** about the order in which the messages sent by S are received by R relative to when they are received by Q

In practice, pretty much all the Erlang code we will write does **not rely on any assumptions** about message delivery order

Even defining – let alone enforcing – an absolute time across multiple independent processes (which could even be geographically distributed) would be tricky: in order to synchronize, processes can only exchange messages!

# Message delivery order: single process

If **process S** sends messages a,b,c – in this order – to **process R**, then **R** will receive them in its mailbox in the **same order**

sender process S:

R ! a,  
 R ! b,  
 R ! c.

receiver process R:

R's mailbox: 

a	b	c
---	---	---

R is process R's pid



# Message delivery order: multiple processes

If **process S** sends messages a,b,c – in this order – to **process R** and to **process Q**, then **R** and **Q** may receive them in **any order** relative to each other.

Possible scenarios:

sender process S:

R ! a,

Q ! b,

Q ! c.

Q is **process Q's pid**



receiver process R:

R's mailbox: a

receiver process Q:

Q's mailbox: b c

# Stateful processes

# A ping server

A **ping server** is constantly listening for requests; to every message `ping`, it replies with a message `ack` sent back to the sender.

In order to **identify the sender**, it is customary to encode messages as tuples of the form:

```
{SenderId, Message}
```

```
ping() -> receive
  {From, ping} -> From ! {self(), ack}; % send ack to pinger
  _           -> ignore                % ignore any other message
end, ping().                             % next message
```

## Combining the echo and ping servers:

```
1> Ping = spawn(echo, ping, []), Echo = spawn(echo, repeat_echo, []).
2> Ping ! {Echo, ping}.      % send ping on Echo's behalf
Received: {<0.64.0>, ack}   % ack printed by Echo
3> Ping ! {Echo, other}.    % send other message to Ping
% no response
```

# Stateful processes

Processes can only operate on the arguments of the function they run, and on whatever is sent to them via message passing

- Thus, we store **state** information using **arguments**, whose value gets updated by the **recursive calls** used to make a process permanently running

A **stateful process** can implement the message-passing analogue of the **concurrent counter** that used Java threads

The Erlang counter function recognizes two commands, sent as messages:

- increment: add one to the stored value
- count: send back the currently stored value

```
base_counter(N) ->
  receive {From, Command} -> case Command of
    increment -> base_counter(N+1);           % increment counter
    count      -> From ! {self(), N},         % send current value
              base_counter(N);               % do not change value
  U          -> io:format("? ~p~n", [U])      % unrecognized command
  end end.
```



# Concurrent counter: first attempt

```

base_counter(N) ->
  receive {From, Command} -> case Command of
    increment -> base_counter(N+1);           % increment counter
    count      -> From ! {self(), N},        % send current value
              base_counter(N);              % do not change value
    U          -> io:format("? ~p~n", [U])    % unrecognized command
  end end.
  
```

Evaluated only when spawning a process running FCount

```

increment_twice() ->
  Counter = spawn(counter, base_counter, [0]), % counter initially 0
  % function sending message 'increment' to Counter:
  FCount = fun () -> Counter ! {self(), increment} end,
  spawn(FCount), spawn(FCount),              % two procs running FCount
  Counter ! {self(), count},                  % send message 'count'
  % wait for response from Counter and print it
  receive {Counter, N} -> io:format("Counter is: ~p~n", [N])
end.
  
```

# Concurrent counter: first attempt (cont'd)

Running `increment_twice` does not seem to behave as expected:

```
1> increment_twice().
Counter is: 0
```

The problem is that there is **no guarantee** that the **message delivery order** is the same as the sending order: the request for `count` may be delivered before the two requests for `increment` (or even before the two processes have sent their `increment` requests).

A temporary workaround is **waiting some time** before asking for the `count`, hoping that the two `increment` messages have been delivered:

```
wait_and_hope() ->
  Counter = spawn(counter, base_counter, [0]),           % counter initially 0
  FCount = fun () -> Counter ! {self(), increment} end,
  spawn(FCount), spawn(FCount),                         % two processes running FCount
  timer:sleep(100),                                     % wait for 'increment' 2b delivered
  Counter ! {self(), count},                             % send message 'count'
  receive {Counter, N} -> io:format("Counter is: ~p~n", [N])
end.
```

# Synchronization in an asynchronous world

Since there is **no guarantee** that the **message delivery order** is the same as the sending order when multiple processes are involved, the only robust mechanism for synchronization is **exchanging messages** following a suitable **protocol**

For example, the counter sends **notifications** of every update to a monitoring process:

```
counter(N, Log) ->
  receive
    {-, increment} ->
      Log ! {self(), N+1}, % send notification
      counter(N+1, Log);   % update count
    {From, count} ->
      % send count, next message
      From ! {self(), N}, counter(N, Log)
  end.
```

# Concurrent counter with monitoring process

```
counter(N, Log) ->
```

```
receive
```

```
{-, increment} -> Log ! {self(), N+1},           % send notification
                  counter(N+1, Log);             % update count
{From, count} -> From ! {self(), N}, counter(N, Log) % send count, next message
end.
```

```
% set up counter and incrementers; then start monitor:
```

```
Increment_and_monitor() ->
```

```
Counter = spawn(?MODULE, counter, [0, self()]),
FCount = fun () -> Counter ! {self(), increment} end,
spawn(FCount), spawn(FCount),
monitor_counter(Counter).           % start monitor
```

Spawns a process from this module and executes function `counter` with parameters `N=0` and `Log=self()` (PID of the spawned process)

`FCount` sends a message to the recently created process (`Counter`) with `self` as parameter and a call to `increment`

```
monitor_counter(Counter) ->
```

```
receive
```

```
{Counter, N} => io:format("Counter is: ~p~n", [N])
```

```
end,
```

```
monitor_counter(Counter).
```

What happens to messages **not** in this format?

They stay in the mailbox!

In the shell: `counter:increment_and_monitor().`

You will get: `Counter is: 1`

`Counter is: 2`

# Quiz

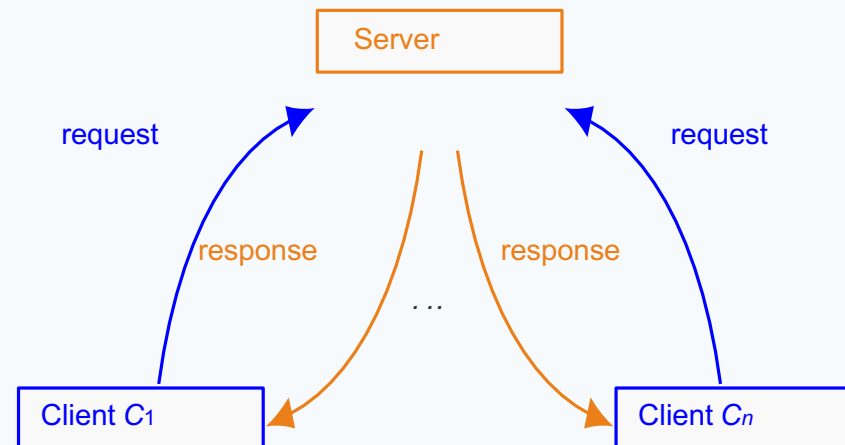
## Erlang and Message Passing

# Clients and servers

# Client/server communication

The **client/server architecture** is a widely used communication model between processes using message passing:

1. A **server** is available to serve requests from any clients
2. An arbitrary number of **clients** send commands to the server and wait for the server's response



Many **Internet** services (the web, email, . . . ) use the client/server architecture

# Servers

A **server** is a process that:

- responds to a fixed number of **commands** – its **interface**
- runs **indefinitely**, serving an arbitrary number of **requests**, until it receives a shutdown command
- can serve an **arbitrary** number of **clients** – which issue commands as **messages**

Each command is a **message** of the form:

`{Command, From, Ref, Arg1, ..., Argn}`

- **Command** is the command's name
- **From** is the pid of the client issuing the command
- **Ref** is a unique identifier of the request (so that clients can match responses to requests)
- **Arg1, ..., Argn** are arguments to the command

Each command is **encapsulated in a function**, so that clients need not know the structure of messages to issue commands



# A math server

The **interface** of a **math server** consists of the following **commands**:

**factorial**(M): compute the factorial of M

**status**() : return the number of requests served so far (without incrementing it)

**stop**() : shutdown the server

We build an Erlang **module** with interface:

**start**() : start a math server, and return the server's pid

**factorial**(S,M): compute factorial of M on server with pid S

**status**(S): return number of requests served by server with pid S

**stop**(S): shutdown server with pid S

```
-module(math_server).
```

```
-export([start/0, factorial/2, status/1, stop/1]).
```

# Math server: event loop

`loop(N) ->`

**receive**

*% 'factorial' command:*

`{factorial, From, Ref, M} ->`

`From ! {response, Ref, compute_factorial(M)},`

`loop(N+1);`

*% increment request number*

*% 'status' command:*

`{status, From, Ref} ->`

`From ! {response, Ref, N},`

`loop(N);`

*% don't increment request number*

*% 'stop' command:*

`{stop, _From, _Ref} -> ok`

**end.**

Ordinary Erlang function computing factorial



This function needs **not** be exported, unless it is spawned by another function of the module using `spawn(?MODULE, loop, [0])`

(In that case, it's called via its module, so it must be exported)

# Math server: starting and stopping

We start the server by spawning a process running `loop(0)`:

```
% start a server, return server's pid  
start() ->  
    spawn(fun () -> loop(0) end).
```

We shutdown the server by sending a command `stop`:

```
% shutdown 'Server'  
stop(Server) ->  
    Server ! {stop, self(), 0}, % Ref is not needed  
    ok.
```

# Math server: factorial and status

We compute a factorial by sending a command `factorial`:

```
% compute factorial(M) on 'Server':
factorial(Server, M) ->
  Ref = make_ref(), % unique reference number
  % send request to server:
  Server ! {factorial, self(), Ref, M},
  % wait for response, and return it:
  receive {response, Ref, Result} -> Result end.
```

Returns a number that is unique among connected nodes in the system

pid of process calling factorial

We get the server's status by sending a command `status`:

```
% return number of requests served so far by 'Server':
status(Server) ->
  Ref = make_ref(), % unique reference number
  % send request to server:
  Server ! {status, self(), Ref},
  % wait for response, and return it:
  receive {response, Ref, Result} -> Result end.
```

# Math server: clients

After creating a server instance, clients simply interact with the server by calling functions of module `math-server`:

```
1> Server = math_server:start().  
<0.27.0>  
2> math_server:factorial(Server, 12).  
479001600  
3> math_server:factorial(Server, 4).  
24  
4> math_server:status(Server).  
2  
5> math_server:status(Server).  
2  
  
5> math_server:stop(Server). ok  
6> math_server:status(Server).  
% blocks waiting for response
```

# Generic servers

# Generic servers

A **generic server** takes care of the communication patterns behind every server

Users instantiate a generic server by providing a suitable **handling function**, which implements a specific server functionality

A generic server's **start** and **stop** functions are almost identical to the math server's – the only difference is that the event loop also includes a handling function:

```
start(InitialState, Handler) ->
  spawn(fun () -> loop(InitialState, Handler) end).
```

Used to receive the “concrete” server implementation we want this generic server to instantiate

```
stop(Server) ->
  Server ! {stop, self(), 0}, % Ref is not needed
  ok.
```

Handler is a function that implements, e.g., all the different operations a Math server might do

# Generic servers: event loop

The generic server's **event loop** has its current state and the handling function as arguments:

```
Loop(State, Handler) ->
```

```
  receive
```

```
    % a request from 'From' with data 'Request'
```

```
    {request, From, Ref, Request} ->
```

```
      % run handler on request
```

```
      case Handler(State, Request) of
```

```
        % get handler's output
```

```
        {reply, NewState, Result} ->
```

```
          % the requester gets the result
```

```
          From ! {response, Ref, Result},
```

```
          % the server continues with the new state
```

```
          Loop(NewState, Handler)
```

```
      end;
```

```
    {stop, _From, _Ref} -> ok
```

```
end.
```



# Generic servers: issuing a request

A generic server's function `request` takes care of sending **generic requests** to the server, and of receiving back the results:

```
% issue a request to 'Server'; return answer  
request(Server, Request) ->  
  Ref = make_ref(), % unique reference number  
  % send request to server  
  Server ! {request, self(), Ref, Request},  
  % wait for response, and return it  
  receive {response, Ref, Result} -> Result end.
```

# Math server: using the generic server

Here is how we can define the `math server` using the `generic` server (starting and stopping use the handling function `math—handler`):

```
start() -> gserver:start(0, fun math—handler/2).
```

```
stop(Server) -> gserver:stop(Server).
```

The handling function has two cases, one per request kind:

```
math—handler(N, {factorial, M}) -> {reply, N+1, compute—factorial(M)};
```

```
math—handler(N, status) -> {reply, N, N}.
```

The exported functions `factorial` and `status` (called by clients) call the generic server's request function:

```
factorial(Server, M) -> gserver:request(Server, {factorial, M}).
```

```
status(Server) -> gserver:request(Server, status).
```

# Servers: improving robustness and flexibility

We extend the implementation of the generic server to **improve**:

**robustness:** add support for error handling and crashes

**flexibility:** add support for updating the server's functionality while the server is running

**performance:** discard spurious messages sent to the server, getting rid of “junk” in the mailbox

All these extensions to the generic server do not change its **interface**

- Thus instance servers relying on it will still work, with the **added benefits** provided by the new functionality!

# Robust servers

If computing the **handling function** on the input **fails**, we **catch** the resulting exception and notify the client that an error has occurred

To handle any possible exception, use the **catch(E)** built-in function:

if evaluating **E** succeeds, the result is propagated;

if evaluating **E** fails, the resulting exception **Reason** is propagated as `{'EXIT', Reason}`

This is how we perform **exception handling** in the **event loop**:

```
case catch(Handler(State, Request)) of
  % in case of error
  {'EXIT', Reason} ->
    % the requester gets the exception
    From ! {error, Ref, Reason},
    % the server continues in the same state
    Loop(State, Handler);
  % otherwise (no error): get handler's output
  {reply, NewState, Result} ->
```

# Flexible servers

Changing the server's functionality requires a new **kind** of **request**, which does not change the server's state but it **changes the handling function**

The event loop now receives also this new request kind:

```
% a request to swap 'NewHandler' for 'Handler'
{update, From, Ref, NewHandler} ->
  From ! {ok, Ref}, % ack
  % the server continues with the new handler
loop(State, NewHandler);
```

Function `update` takes care of sending requests for changing handling function (similarly to what `request` does for basic requests):

```
% change 'Server's handler to 'NewHandler'
update(Server, NewHandler) ->
  Ref = make_ref(), % send update request to server
  Server ! {update, self(), Ref, NewHandler},
  receive {ok, Ref} -> ok end. % wait for ack
```

← Allows for “hot upgrading”

# Discarding junk messages

If **unrecognized messages** are sent to a server, they remain in the mailbox indefinitely (they never pattern match in **receive**)

If too many such “junk” messages pile up in the mailbox, they may **slow down** the server

To avoid this, it is sufficient to match any unknown messages and discard them as last clause in the event loop’s **receive**:

```
% discard unrecognized messages  
- -> loop(State, Handler)
```

To avoid clients waiting forever for responses to discarded requests, we add a **timeout** to request:

```
receive  
{response, Ref, Result} -> Result  
% after 10 seconds, give up  
after 10000 -> timeout end.
```

# Location transparency and distribution

# Registered processes

One needs another process's pid to exchange messages with it. To increase the flexibility of **exchanging pids** in open systems, it is possible to **register** processes with a symbolic name:

- **register**(Name, Pid): register the process Pid under Name; from now on, Name can be used wherever a pid is required
- **unregister**(Name): unregister the process under Name; when a registered process terminates, it implicitly unregisters as well
- **registered**(): list all names of registered processes
- **whereis**(Name): return pid registered under Name

In the **generic server**, we can add a registration function with name:

```
% start a server and register with 'Name'  
start(InitialState, Handler, Name) ->  
    register(Name, start(InitialState, Handler)).
```

All other server functions can be used by passing Name for Server



# From concurrent to distributed

Message passing concurrency works in the same way independent of whether the processes run on the same computer or in a **distributed setting**

In Erlang, we can turn any application into a distributed one by running processes on **different nodes**:

- start an Erlang runtime environment on each node
- connect the nodes by issuing a ping
- load the modules to be execute on all nodes in the cluster
- for convenience, register the server processes
- to identify registered process `Name` running on a node `node@net-address` use the tuple `{Name, 'node@net-address'}` wherever you would normally use a registered name or pid

# Distribution: setting up nodes


In our simple experiments, the nodes are processes on the same physical local machine (IP address `127.0.0.1`, a.k.a. local host), but the very same commands work on different machines connected by a network

Node server@127.0.0.1:

```
> erl -name 'server@127.0.0.1'
    -setcookie math_cluster
s1>
```

Node client@127.0.0.1:

```
> erl -name 'client@127.0.0.1'
    -setcookie math_cluster
c1>
```



By using the flag `setcookie` we give a symbolic name that all the nodes in a group share (only those nodes having the same cookie can interact - to avoid unwanted connections from processes in other nodes)

(A cookie is an identifier that all nodes in the same connected group share)

# Distribution: connect nodes and load modules

Nodes are invisible to each other until a message is exchanged between them; after that, they are **connected**

## Node client@127.0.0.1:

*% send a ping message to connect client to server node*

```
c1> net-adm:ping('server@127.0.0.1').
```

```
pong % the nodes are now connected
```

*% list connected nodes*

```
c2> nodes() .
```

```
['server@127.0.0.1']
```

*% load module 'ms' in all connected nodes*

```
c3> n1(ms) .
```

```
abcast % the module is now loaded
```

# Distribution: server setup

We **start the math server** on the node `server` and register it under the name `mserver`. Then, we can **issue request** from the client node using

`{mserver, 'server@127.0.0.1'}` instead of `pids`.

## Node server@127.0.0.1:

```
s1> register(mserver, ms:start()).  
true  
% server started  
% and registered
```

## Node client@127.0.0.1:

```
c4> ms:factorial({mserver, 'server@127.0.0.1'}, 10).  
3628800  
c5> ms:status({mserver, 'server@127.0.0.1'}).  
1  
c6> ms:status({mserver, 'server@127.0.0.1'}).  
1
```

The very same protocol works for an arbitrary number of client nodes

# These slides' licence

© 2016–2019 Carlo A. Furia, Sandro Stucki



Except where otherwise noted, this work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/4.0/>.