Introduction to Erlang processes and message passing

Module 8 - Erlang tutorial 3

Concurrency ≠ Parallelism

Processes

Message passing

Operating systems and process oriented programming 2020

1DT096

Spring 2020

karl.marklund@it.uu.se

Uppsala university

Concurrency

≠

Parallelism

Concurrency ≠ Parallelism

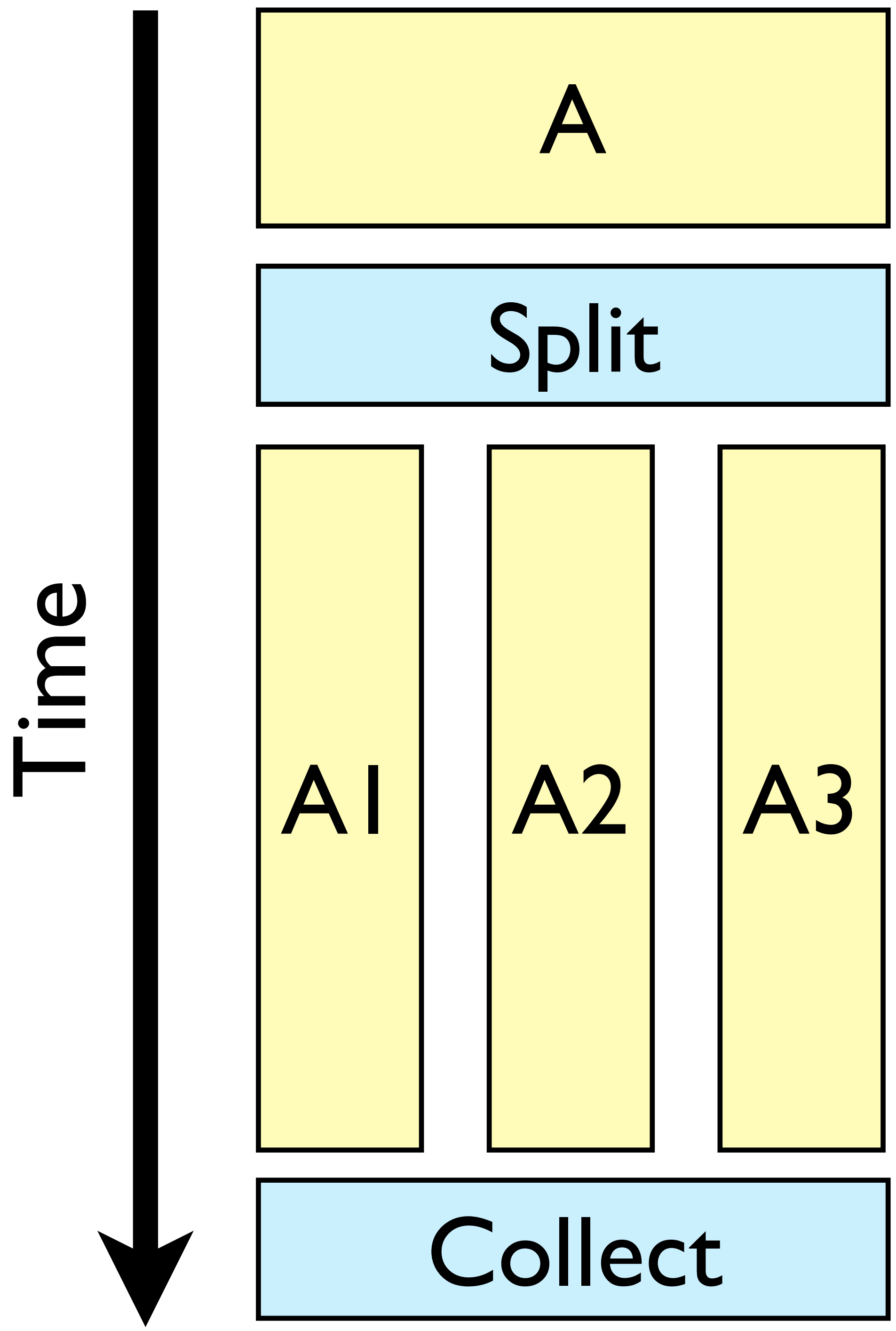
In **parallel system** tasks are actually performed simultaneously.

Parallelism is when tasks literally run at the same time, eg. on a multicore processor.

Concurrency is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. Eg. multitasking on a single-core machine.

Concurrent systems give the appearance of several tasks executing at once, but these tasks are actually split up into chunks that share a resource (e.g. the processor or a server) with chunks from other tasks by interleaving the execution in a time-slicing way.

In a concurrent system, concurrent tasks *may* be executed in parallel (if the hardware allows) but can also be executed on a single processor by interleaving the execution steps of each in a time-slicing way, Parallel



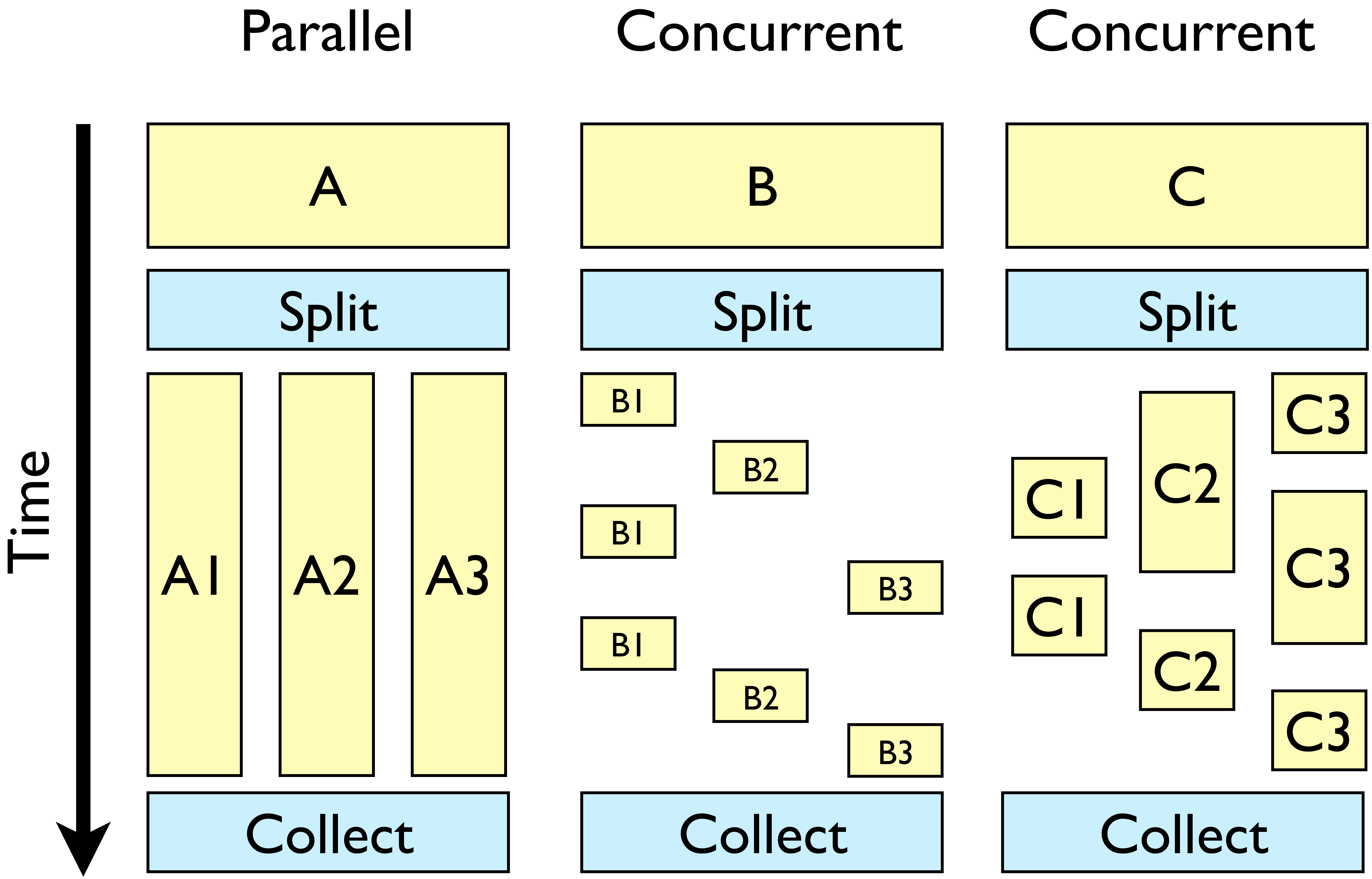
Example

A pile of dirty clothes

Sorted into three piles by color.

Washed in parallel in three machines.

A pile of clean clothes.


Parallel
Concurrent
Concurrent
Time
A
B
C
Split
Split
Split
A1
A2
A3
B1
B2
B1
B3
B1
B2
B3
C3
C2
C1
C3
C1
C2
C3
Collect
Collect
Collect

WEDNESDAY, AUGUST 16, 2006

Why Erlang Is a Great Language for Concurrent Programming

ABOUT ME

YARIV

VIEW MY COMPLETE PROFILE

ABOUT ME

I work at Facebook but the opinions I publish in this blog are my own.

One of the greatest aspects of Erlang's concurrency is the way it's implemented behind the scenes.

Erlang code runs in a **virtual machine** called BEAM, which is responsible for **spawning**, **scheduling**, and **cleaning up** Erlang **processes**, as well as **passing messages** between them.

Erlang processes are much more **lightweight** than OS threads, which means that you can **potentially spawn millions of processes on a single box**. Try that with a Pthread-based library and your machine would croak after the first 4000-10000 threads.

Erlang also maps nicely onto multi-core CPUs - why is this? - precisely because we use **a non-shared lots of parallel processes model of computation**.

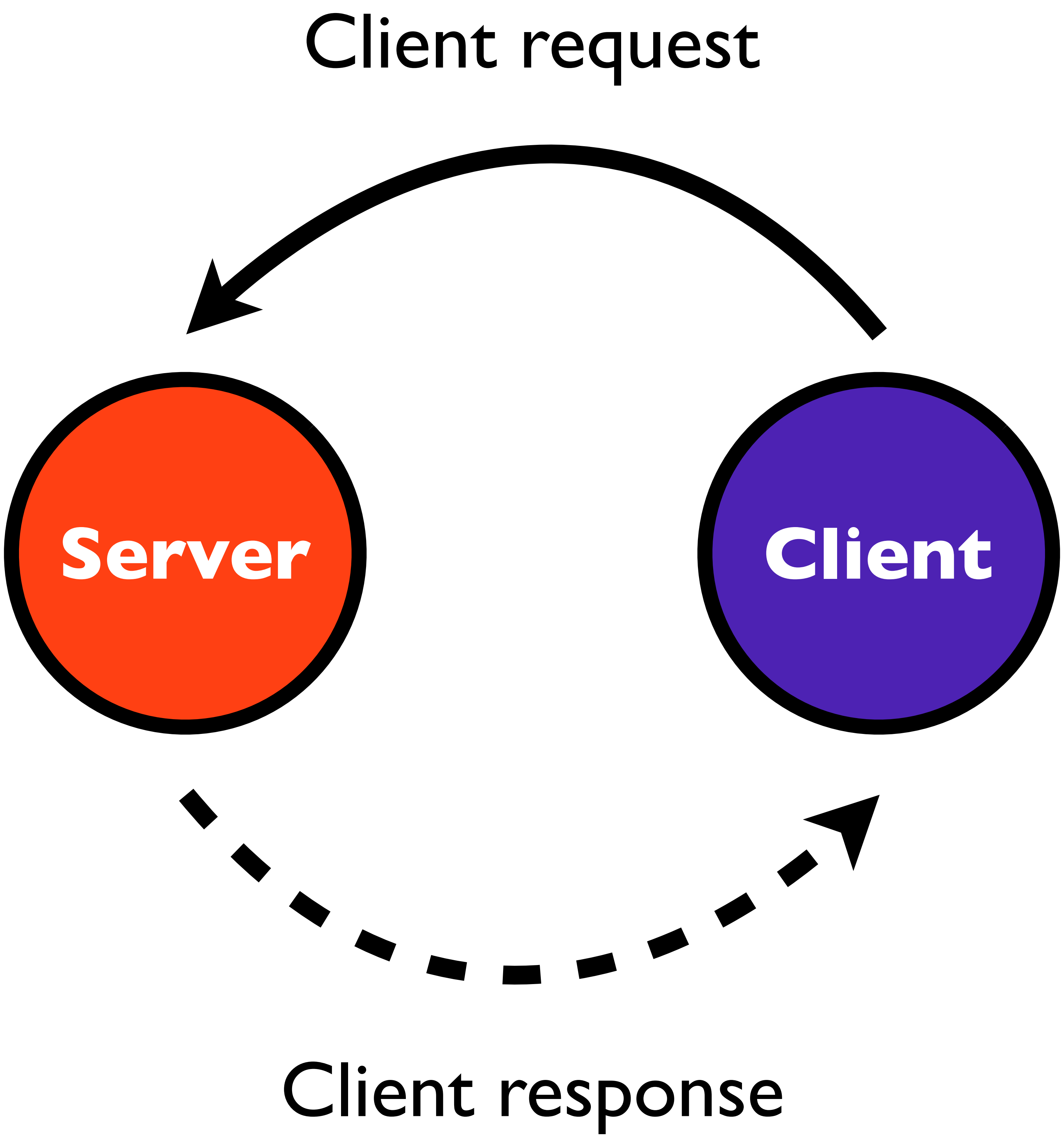
- No shared memory
- No threads
- No locks

➡

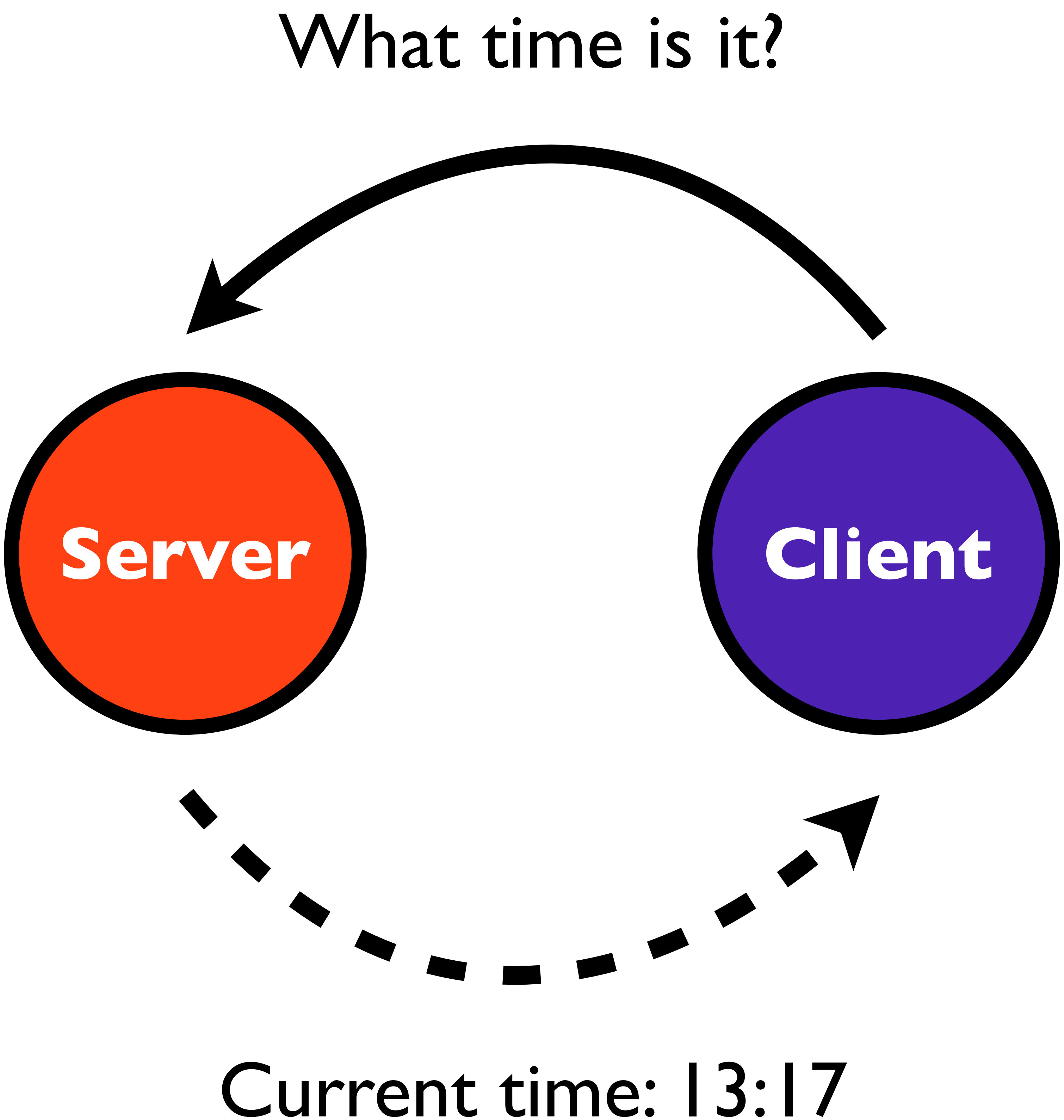
Ease of running on a parallel CPU.

Clients and servers

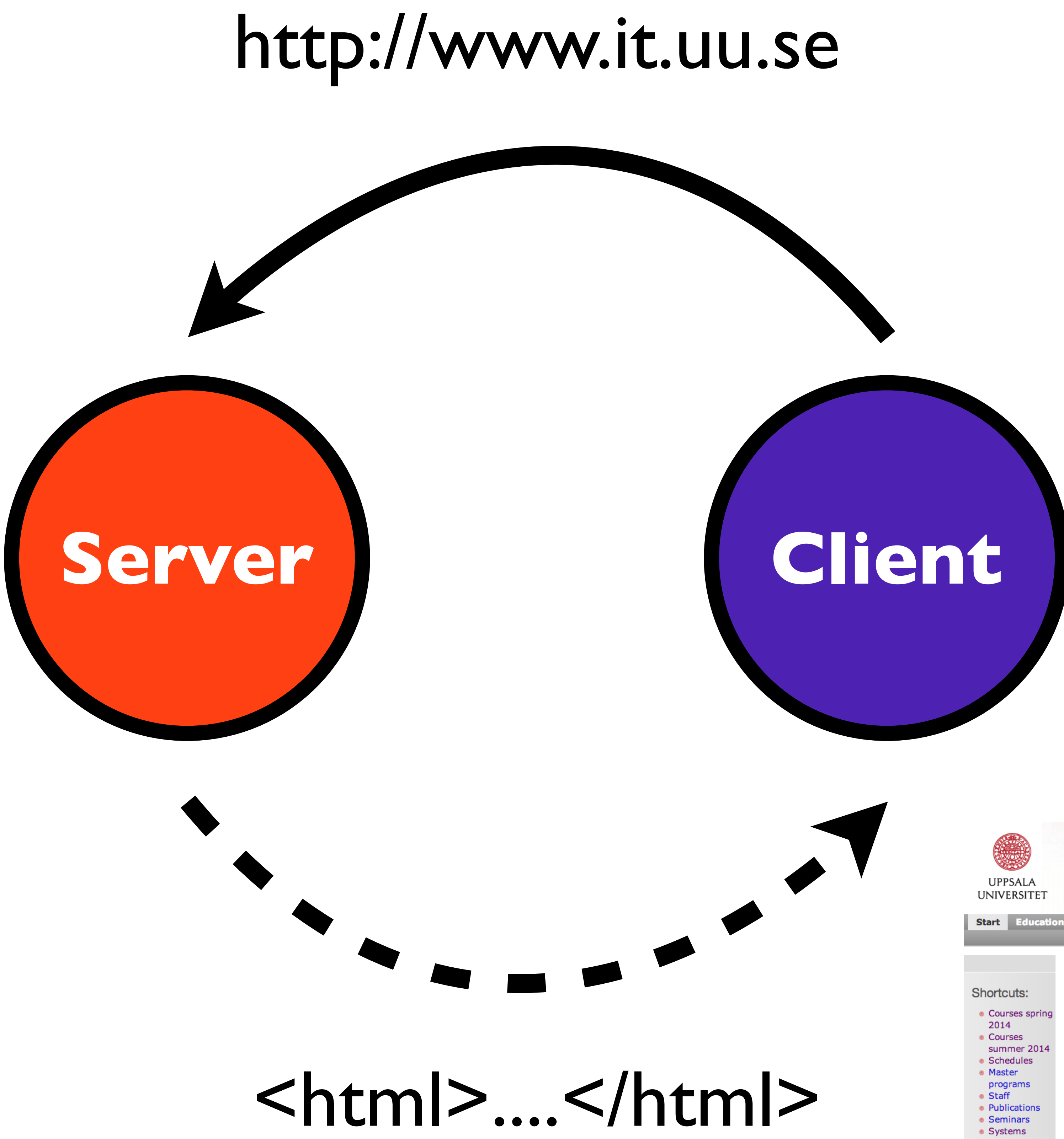
A computer network (for example the Internet)

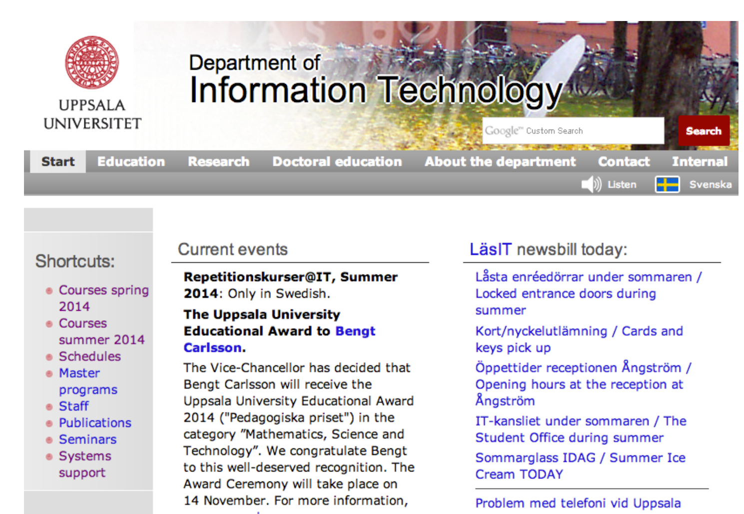


A computer network (for example the Internet)

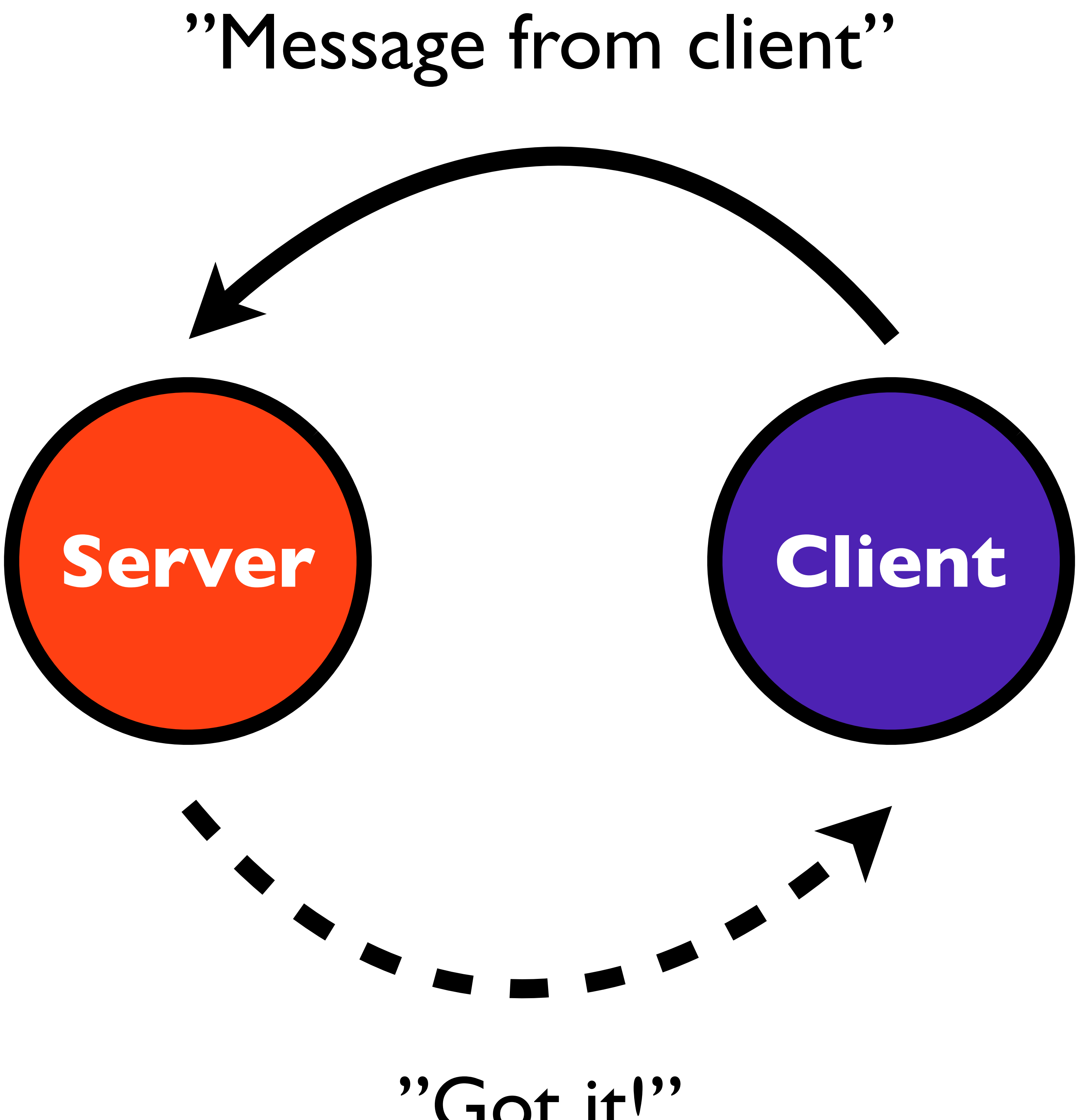


A web browser is a HTML client sending requests to a web server.

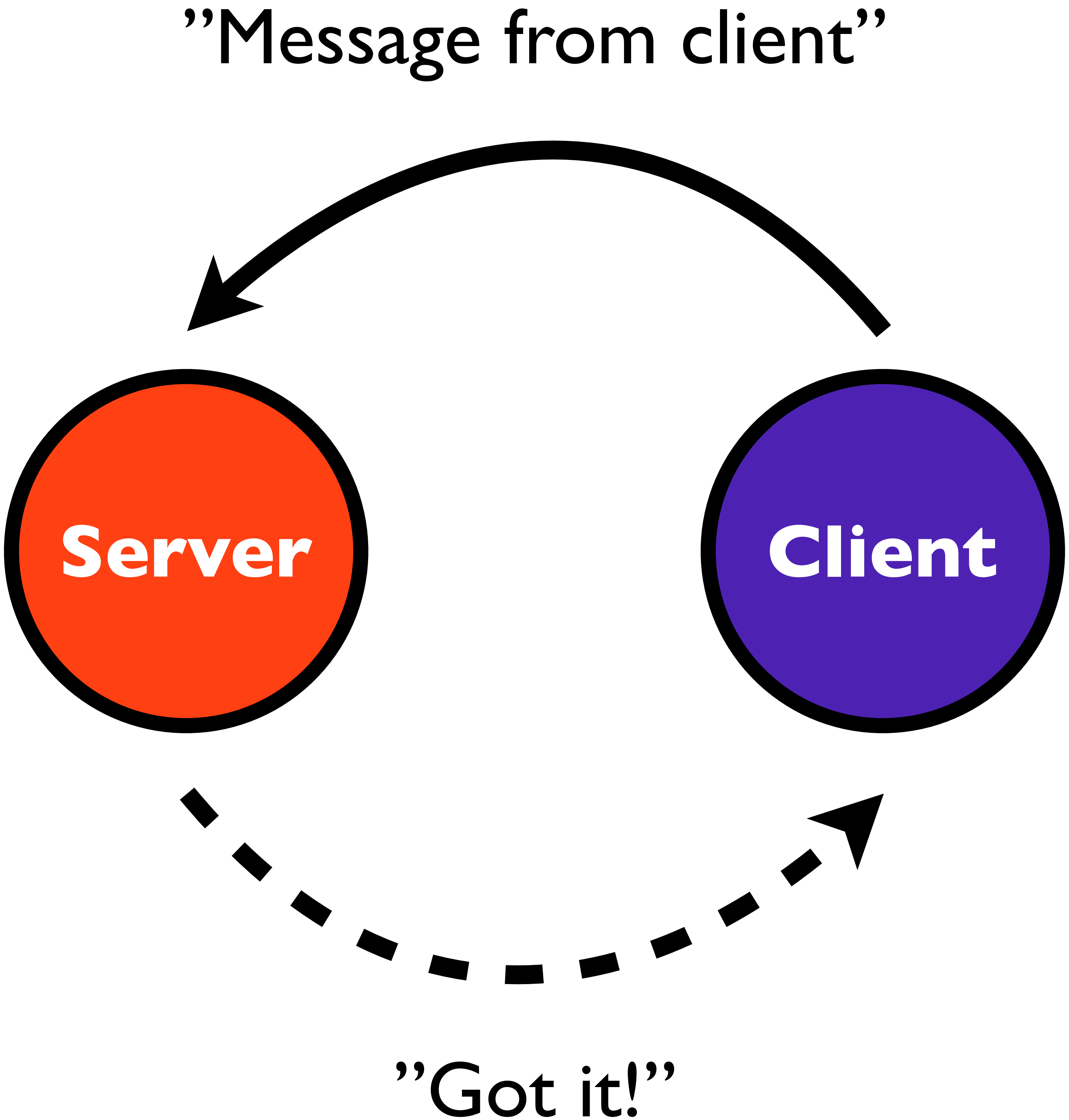




A simple example



Let's try to implement this in Java



Clients and servers in Java

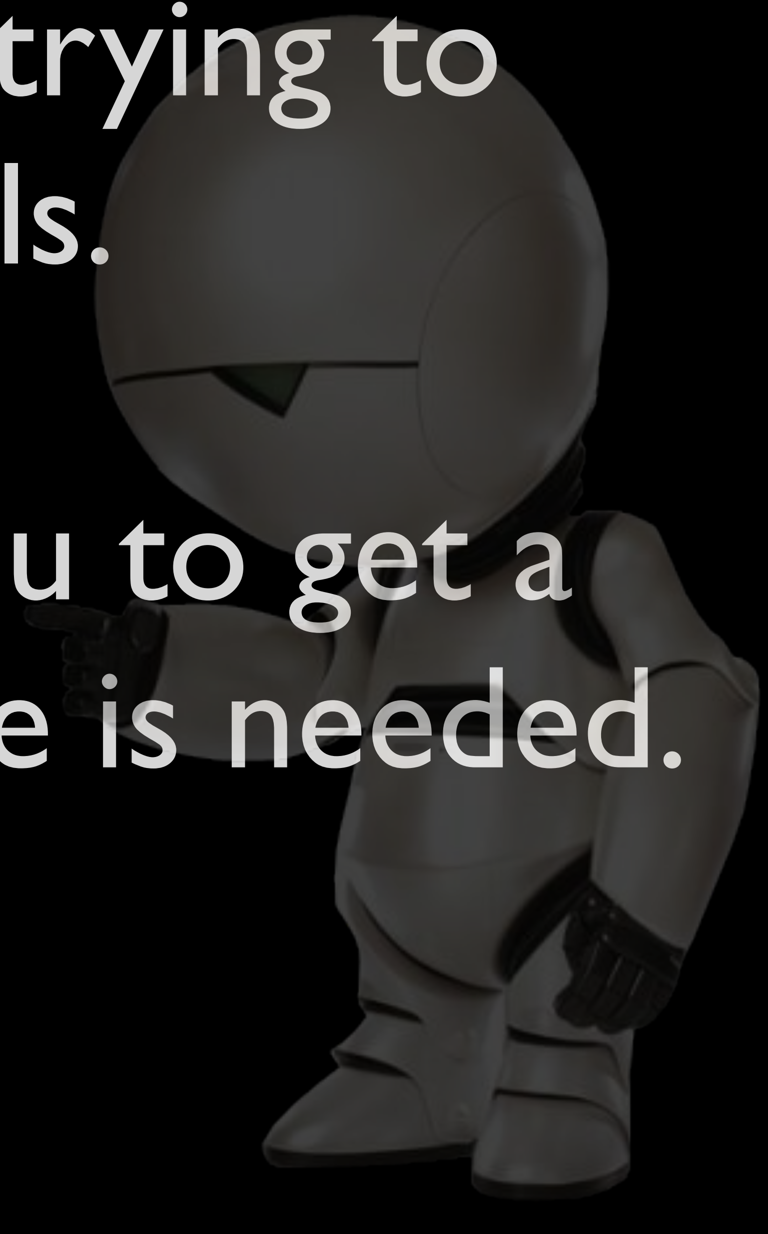
DON'T

PANIC

The next two slides will show how to implement the example client-server system in the Java programming language.

Don’t spend to much time trying to understand the details.

The purpose is simply for you to get a feeling for how much Java code is needed.



Server

```java
public class Server {
  final static int serverPort = 3456;  // Server port number.

  public static void main(String args[]) {

    java.net.ServerSocket  sock         = null;  // Original server socket.
    java.net.Socket        clientSocket = null;  // Socket created by accept.
    java.io.PrintWriter    pw           = null;  // Socket output stream.
    java.io.BufferedReader br           = null;  // Socket input stream.

    try {
      // Create socket and bind to port.
      sock = new java.net.ServerSocket(serverPort);

      // Wait for client to connect.
      System.out.println("waiting for client to connect");
      clientSocket = sock.accept();
      System.out.println("client has connected");

      pw = new java.io.PrintWriter(clientSocket.getOutputStream(), true);
      br = new java.io.BufferedReader(new java.io.InputStreamReader(clientSocket.getInputStream()));

      // Read message from client.
      String msg = br.readLine();
      System.out.println("Message from the client >" + msg);

      // Send message to client.
      pw.println("Got it!");

      // Close everything.
      pw.close();
      br.close();
      clientSocket.close();
      sock.close();

    } catch (Throwable e) {
      System.out.println("Error " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

The **server** uses a TCP socket to receive a message from the client and send a response.

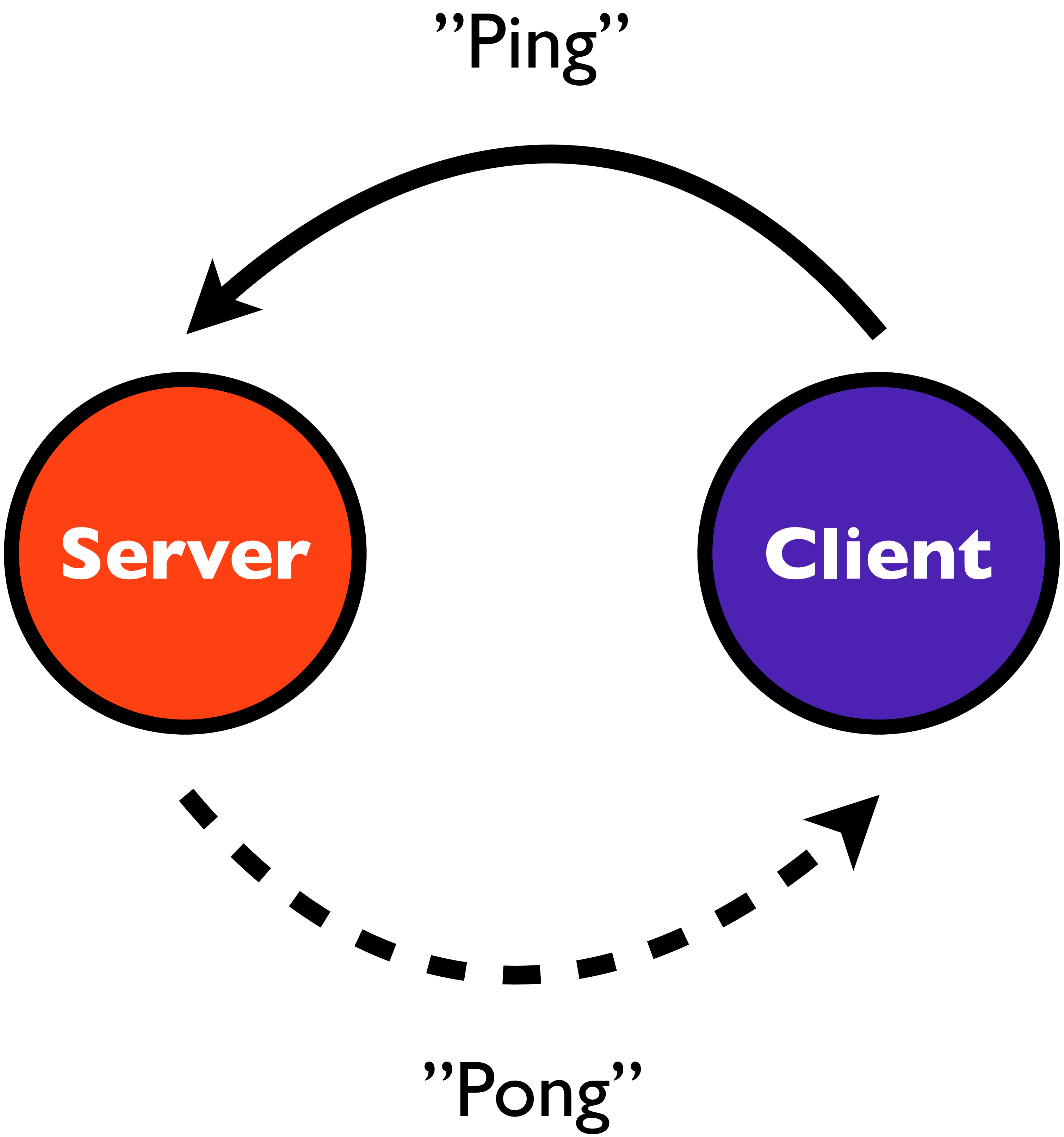
Client

```java
public class Client {
  final static String serverIPname = "hamberg.it.uu.se";
  final static int serverPort      = 3456;

  public static void main(String args[]) {

    java.net.Socket         sock = null; // Socket object for communicating.
    java.io.PrintWriter     pw   = null; // Socket output to server.
    java.io.BufferedReader br    = null; // Socket input from server.

    try {
      // Create socket and connect.
      sock = new java.net.Socket(serverIPname,serverPort);
      pw   = new java.io.PrintWriter(sock.getOutputStream(), true);
      br = new java.io.BufferedReader(new java.io.InputStreamReader(sock.getInputStream()));
      System.out.println("Connected to Server");

      // Send message to the server.
      pw.println("Message from the client");
      System.out.println("Sent message to server");

      // Get data from the server.
      String answer = br.readLine();
      System.out.println("Response from the server > " + answer);

      // Close everything.
      pw.close();
      br.close();
      sock.close();

    } catch (Throwable e) {
      System.out.println("Error " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

The **client** uses a TCP socket to send a message to the server and receive a response.

Clients and servers in pseudo code

Let’s invent a more dense notation



Sending a message

This is such a common operation, lets make the syntax for sending a message really compact.

```
A ! Message
```

In the above example we send **`Message`** to **`A`**.

We haven’t yet defined what **`A`** is, neither have we defined what a **`Message`** is.

Receiving a message

This is such a common operation, lets make the syntax for receiving a message really compact.

```
receive
    Message -> %% Action
end
```

We use a **`receive ... end`** block to receive a **`Message`**.

When a **`Message`** arrives, the operations following **`->`** (right arrow) will be performed.

Let **%%** denote the beginning of a comment.

Functions

A function takes one or more arguments and returns a result.

```
doulbe(X) -> 2*X
```

In the above example, **`double`** is the name of a function taking one argument named **`X`** and returning the result of multiplying **`X`** by **`2`**.

```
sum(X,Y) -> X+Y
```

The function **`sum`** takes two arguments **`X`** and **`Y and returns`** their sum.

Client and server

```
client()->
   Server ! "Ping"
```

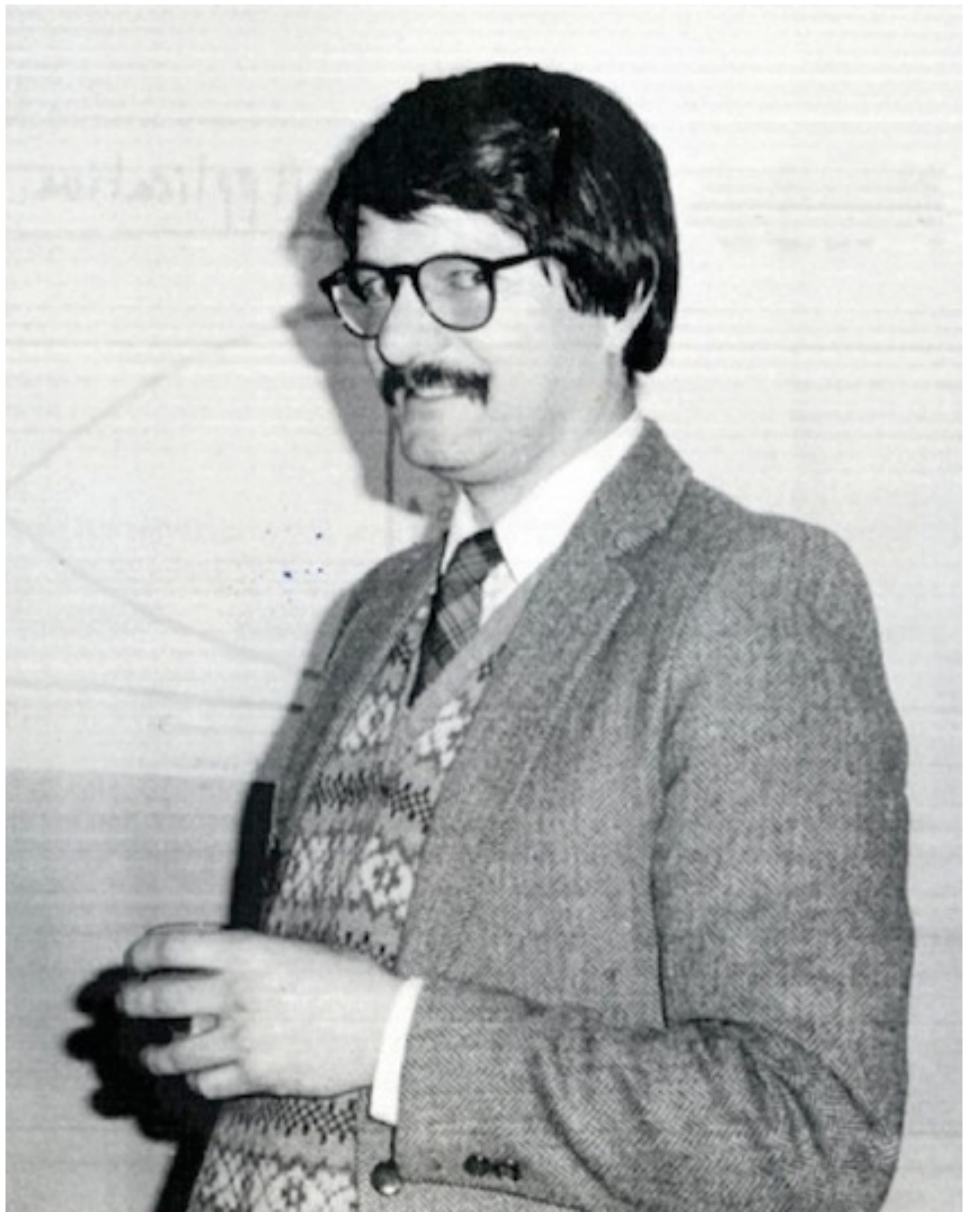
The function **`client`** sends the string **`"Ping"`** to server.

```
server()->
   receive
     Msg -> Client ! "Pong"
   end
```

The function **`server`** waits for a message from the server and replies with the string **`"Pong"`**.

Message passing in Erlang

Time to learn some Erlang



Joe Armstrong
1878 - 1929

A computer scientist working in the area of fault-tolerant distributed systems. He is best known as one of the co-designers of the Erlang programming language.

The first version of the Erlang programming language was developed by Joe Armstrong in 1986. Erlang was originally a proprietary language within Ericsson, but was released as open source in 1998.

The Erlang syntax is based on the same ideas as our experiment with a more dense notation for message passing.

Agner Krarup Erlang
1878 - 1929

A Danish mathematician, statistician and engineer, who invented the fields of traffic engineering and queueing theory.

The Erlang shell

A good way starting to learn Erlang is by playing around in the interactive Erlang emulator, the Erlang shell.

To start the Erlang shell, open a terminal and then type in **`erl`** and press enter.

```
$> erl
```

Here **`$>`** is used to denote the **Linux** shell **prompt**. You may see a different prompt depending on the system you are using.

Erlang concurrent processes

spawn/3

To create a new process, the build in function (**BIF**) **`spawn/3`** can be used.

```
spawn(Module, Function, Args) -> pid()
```

The spawn/3 BIF will spawn a new process running the function Function in module Module with the values in the list Args as arguments. The PID of the new process is returned by spawn/3.

In Erlang lingo, the acronym **MFA** is often used to denote a tuple of the form:

```
{Module, Function, Arguments}
```

The second argument **Function** to spawn/3 must be **exported** from the module Module.

In logic, mathematics, and computer science, the **arity** of a function or operation is the number of arguments or operands that the function takes. In Erlang, this is commonly denoted by appending /n to the name of a function with arity n.

Erlang concurrent processes

spawn/1

To create a new process, the build in function (**BIF**) **`spawn/1`** can be used.

```
spawn(Fun) -> pid()
```

The spawn/1 BIF returns the process identifier of a new process started by the application of Fun to the empty list []. Otherwise works like spawn/3.

Types

Fun = function()

self/0

The built in function (BIF) **`self/0`** returns the Process ID (PID) of the current process.

```
1> self().
<0.41.0>
```

The Erlang shell itself is implemented as a regular process, hence we can get the PID of the shell using **`self/0`**.

The Erlang ”crypto”

`!` `.` `=` `,` `->` `;`

Where does the Erlang syntax come from?

Mostly from Prolog. Erlang started life as a modified Prolog. **!** as the send-message operator comes from CSP. Eripascal was probably responsible for **,** and **;** being separators and not terminators.

!

In Erlang the **bang** operator (!) is used to send a message from one process to another.

```
PID ! Message
```

PID is the receiving process ID and Message can be any valid Erlang term.

In the Erlang shell, the **period** (.) ends an expression. In modules, the period ends forms. Forms are module attributes and function declarations.

Hence, to send a message we must end the expression with a period.

```
PID ! Message.
```

=

In Erlang, the equal sign (=) is called the **match operator**.

In Erlang, a variable can be either **bound** or **unbound**. When used with an unbound variable on the left hand side, the match operator (=) will **bind** the right hand value to the left hand variable.

```
A = 1.
```

When used together with a bound variable, the operator is used for **pattern matching**.

```
A = 1.
A = 2.
exception error: no match of right hand side value 2
```

=

Pattern matching can be used to unpack compound data structures.

```
{A, B, C} = {42, [1,2,3], {foo, bar}}.
```

In the above example, A is bound to integer 42, B to the list [1,2,3] and C to the tuple {foo, bar}.

```
{A, B, C, D} = {42, [1,2,3], {foo, bar}}.
** exception error: no match of right hand side value
        {42,[1,2,3],{foo,bar}}
```

In the above example, a 4 tuple cannot be matched against a 3 tuple.

,

In Erlang the comma (,) separates expressions:

```
C = A + B, D = A + C.
```

->

In Erlang the right arrow **`->`** is used to separate the head of a clause from the body, for example the function head from the function body.

```
double(N) -> 2*N.
```

We have now defined a function named double. Calling double(6) returns 12.

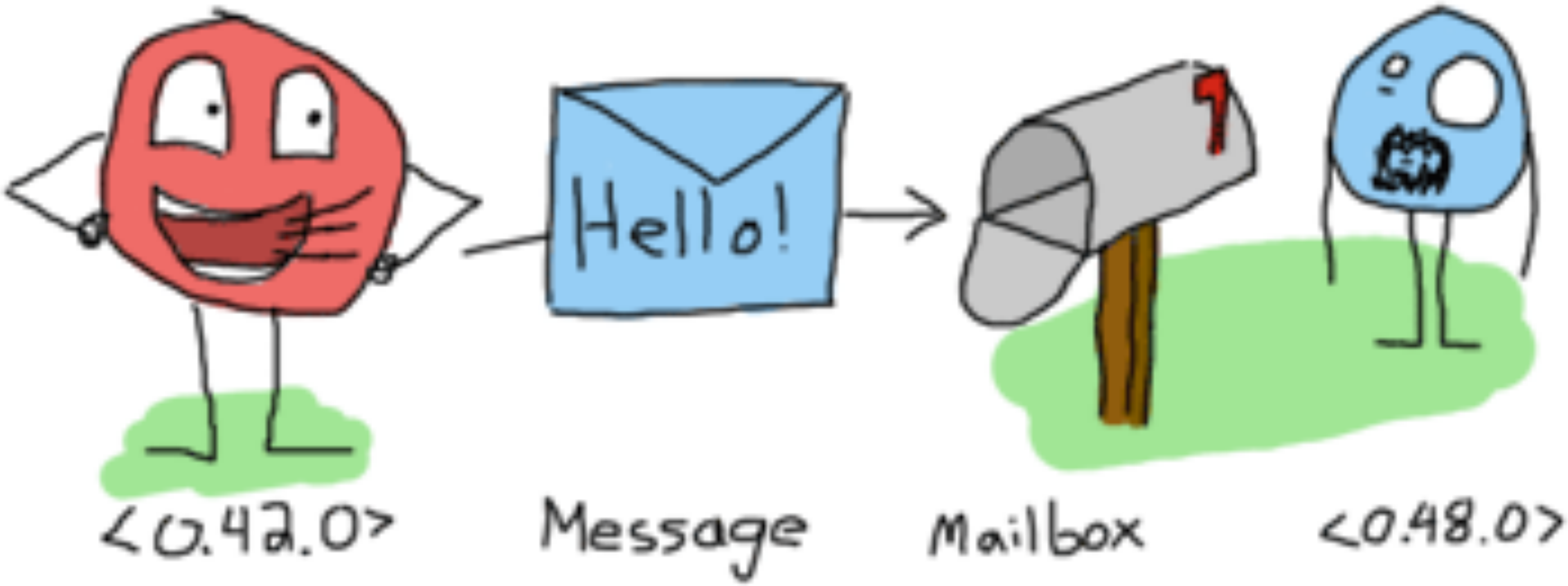
;

In Erlang the semi-colon is used to separate ”choices” or more formally, separate clauses. For example, a function can be defined with multiple clauses, for example:

```
fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

Note that the last function clause must end with a period (.).

Message passing examples in Erlang



To send a message to another process, the sender must know the process id (PID) of the receiver.

Send the atom hello to your own mailbox.

```
2> self() ! hello.
hello
3>
```

The message has been put in the process' mailbox, but it hasn't been read yet. The second hello shown here is the value of the send expression.

The `receive` expression is used to wait for messages to appear in the mailbox.

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

semicolon (;) separates the patterns.

no semicolon after the last pattern

The patterns Pattern are sequentially matched against the first message in time order in the mailbox, then the second, and so on. If a match succeeds and the optional guard sequence GuardSeq is true, the corresponding Body is evaluated. The matching message is consumed, that is removed from the mailbox, while any other messages in the mailbox remain unchanged.

The value of the matching Body expression is the value of the whole receive expression.

We will discuss **guards** in a moment ...

Dolphins

```erlang
-module(dolphins).
-compile(export_all).

dolphin() ->
  receive
    do_a_flip ->
      io:format("How about no?~n"),
      dolphin(); %% Recursive call
    fish ->
      io:format("So long and thanks for all the fish!~n");
    _ ->
      io:format("Heh, we're smarter than you humans.~n"),
      dolphin()  %% Recursive call
  end.
```

Save the following Erlang code in a file named **dolphins.erl**

Inside the receive expression we use **patterns** to describe what kind of messages we are waiting for. If a matching message is found, the action after -> is executed.

In the **dolphin1/0** function we use the **receive ... end** expression to wait for messages to appear in the process mailbox.

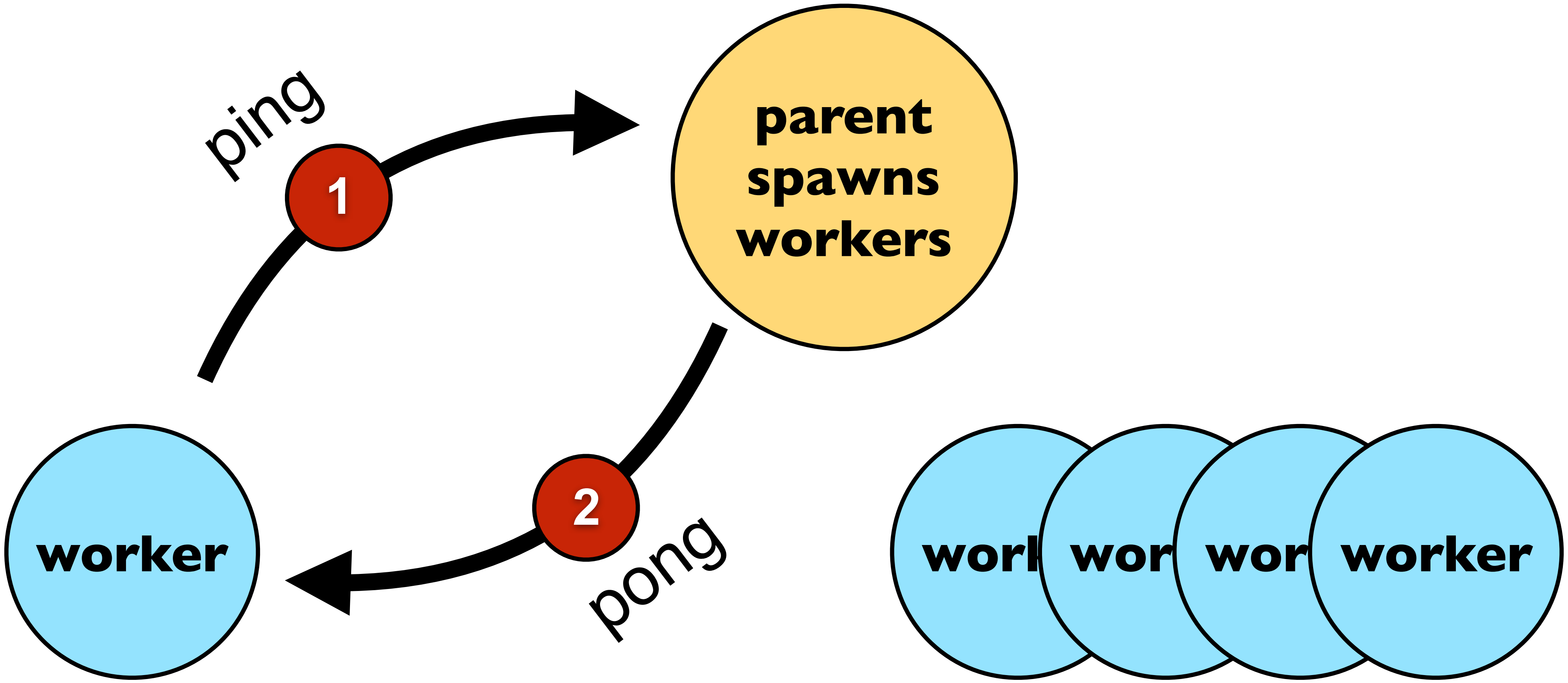
```erlang
-module(dolphins).
-compile(export_all).

dolphin() ->
  receive
    do_a_flip ->
       io:format("How about no?~n"),
       dolphin(); %% Recursive call
    fish ->
       io:format("So long and thanks for all the fish!~n");
    _ ->
       io:format("Heh, we're smarter than you humans.~n"),
       dolphin()  %% Recursive call
  end.
```

```
3> c(dolphins).
{ok,dolphins}
4> Dolph = spawn(dolphins, dolphin, []).
<0.65.0>
5> Dolph ! "hello".
Heh, we're smarter than you humans.
"hello"
6> Dolph ! fish.
So long and thanks for all the fish!
fish
```

Ping Pong

Ping Pong

A parent process spawns a number of worker processes. Each worker sends a ping message to the parent and the parent replies with a pong.



Use a list comprehension to spawn N worker processes.

```
-module(ping).
-export([start/1]).
%% All functions used by spawn/3 must be exported.
-export([worker/1]).

start(N) ->
    io:format("~p Parent process started.~n", [self()]),
    [spawn(?MODULE, worker, [self()]) || _ <- lists:seq(1, N)],
    parent(N).
```

The **?MODULE** macro is often used together with spawn. **?MODULE** is replaced with the current module name by the compiler.

After spawning the worker processes, the parent will execute the following function.

```erlang
parent(0) ->
    io:format("~p Received ping from all workers.~n",
              [self()]);
parent(N) ->
     receive
         {ping, From} ->
             io:format("~p Ping from worker ~p.~n",
                       [self(), From]),
             From ! {pong, self()},
             parent(N-1)
     end.
```

Each worker process will execute the following function.

```
worker(Parent) ->
    io:format("~p Worker started.~n", [self()]),

    Parent ! {ping, self()},

    receive
        {pong, From} ->
            io:format("~p Pong from parent ~p.~n",
                      [self(), From])
    end.
```

```erlang
-module(ping).
-export([start/1]).
%% All functions used by spawn/3 must be exported.
-export([worker/1]).

start(N) ->
    io:format("~p Parent process started.~n", [self()]),
    [spawn(?MODULE, worker, [self()]) || _ <- lists:seq(1, N)],
    parent(N).

parent(0) ->
    io:format("~p Received ping from all workers.~n", [self()]);
parent(N) ->
     receive
         {ping, From} ->
             io:format("~p Ping from worker ~p.~n", [self(), From]),
             From ! {pong, self()},
             parent(N-1)
     end.

worker(Parent) ->
    io:format("~p Worker started.~n", [self()]),

    Parent ! {ping, self()},

    receive
        {pong, From} ->
            io:format("~p Pong from parent ~p.~n", [self(), From])
    end.
```

```
1> c(ping).
{ok,ping}
2> ping:start(5).
<0.32.0> Parent process started.
<0.39.0> Worker started.
<0.40.0> Worker started.
<0.41.0> Worker started.
<0.42.0> Worker started.
<0.43.0> Worker started.
<0.32.0> Ping from worker <0.39.0>.
<0.32.0> Ping from worker <0.40.0>.
<0.39.0> Pong from parent <0.32.0>.
<0.32.0> Ping from worker <0.41.0>.
<0.40.0> Pong from parent <0.32.0>.
<0.32.0> Ping from worker <0.42.0>.
<0.41.0> Pong from parent <0.32.0>.
<0.32.0> Ping from worker <0.43.0>.
<0.42.0> Pong from parent <0.32.0>.
<0.32.0> Received ping from all workers.
<0.43.0> Pong from parent <0.32.0>.
ok
3>
```

Concurrent execution of the parent process and the worker processes.

Sending functions in messages

Sending functions in messages

Any Erlang term can be sent in a message, even functions. Can use the BIF **is_function/2** to tell if a fun of a certain arity is received or not.

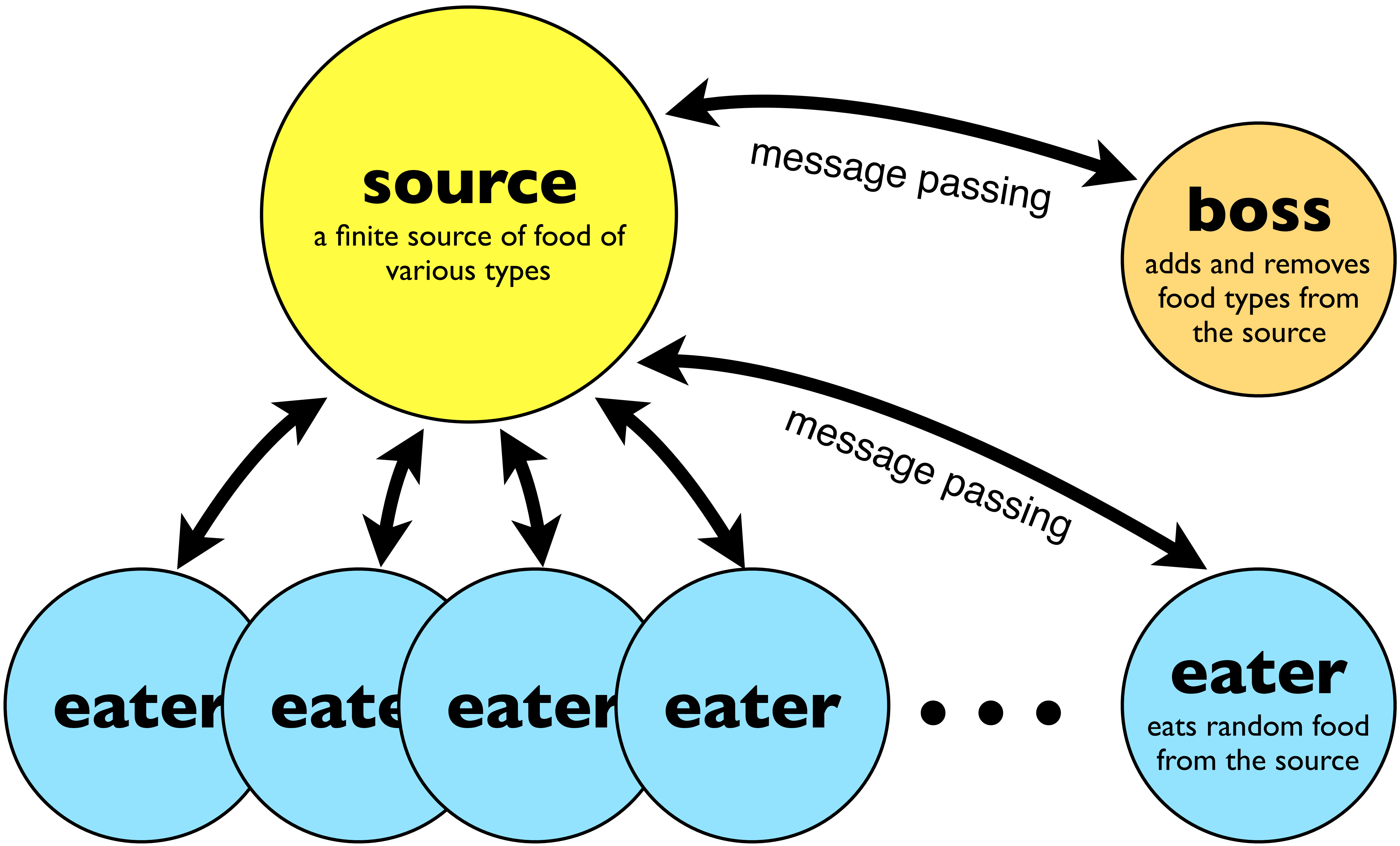
```
-module(funs).
-export([start/0]).

start() ->
    PID = spawn(fun() -> loop(1) end),
    io:format("~w Lets send some messages to ~w~n", [self(), PID]),
    PID ! fun(X) -> 2*X end,
    PID ! foo,
    PID ! fun foo/1,
    PID ! fun(X) -> 3*X end,
    ok.

foo(X) ->
    X*X.

loop(N) ->
    receive
      X when is_atom(X) ->
         io:format("~w atom ~w received~n", [self(), X]),
         loop(N);
      Fun when is_function(Fun,1) ->
         Y = Fun(N),
         io:format("~w Fun(~w) = ~w~n", [self(), N, Y]),
         loop(N+1)
    end.
```

```
11> c(funs).
{ok,funs}
12>
12> funs:start().
<0.86.0> Lets send some messages to <0.125.0>
<0.125.0> Fun(1) = 2
ok
<0.125.0> atom foo received
<0.125.0> Fun(2) = 4
<0.125.0> Fun(3) = 9
```

Food and eaters

Erlang processes (example)

An example where we simulate a food source and a number of eaters. The food source has a finite number of food types. On request, a eater will get a random food type. The boss will remove and add new food types to the food source. The source, eaters and boss will be represented as separate **concurrent processes**.



source

```
init_food(Food) ->
    random:seed(erlang:now()),
    loop(Food).

loop([]) ->
    io:format("~p Alert - no food at all!~n", [self()]),
    receive
        {add, Name} ->
            io:format("~p God news, ~s now availalbe!~n", [self(), Name]),
            loop([Name])
    end;

loop(Food) ->
    receive
        {eat, PID} ->
            PID!lists:nth(random:uniform(length(Food)), Food),
            loop(Food);
        {add, Name} ->
            io:format("~p God news, ~s now availalbe!~n", [self(), Name]),
            loop(lists:append(Food, [Name]));
        {remove, Name} ->
            io:format("~p Bad news, no more ~s :(~n", [self(), Name]),
            loop(lists:delete(Name, Food))
    end.
```

The state of the Erlang PRNG is per-process.

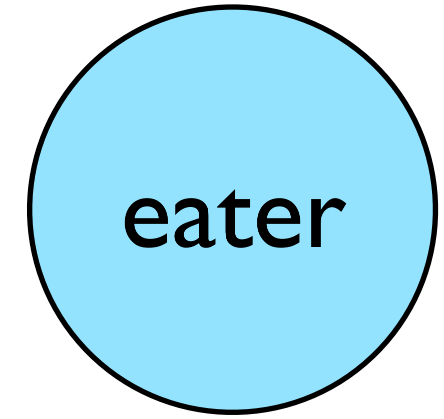
Receive messages using pattern matching.

A recursive call to loop/1

A recursive call to loop/1 with unmodified **state**

A recursive call to loop/1 with a new **state**

A recursive call to loop/1 with a new **state**



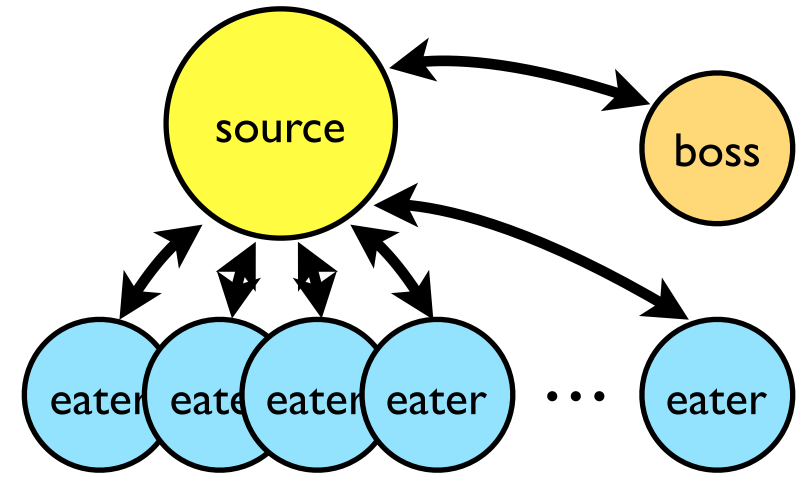

```erlang
eat(Source) ->
    Source!{eat, self()},
    receive
        Food -> ok
    end,
    io:format("~p ~s~n", [self(), Food]).

init_eater(Source, N) ->
    random:seed(erlang:now()),
    eater(Source, N).

eater(_Source, 0) ->
    io:format("~p done~n", [self()]);
eater(Source, N) ->
    timer:sleep(1000 + random:uniform(1000)),
    eat(Source),
    eater(Source, N-1).
```

boss

```erlang
toogle(remove) -> add;
toogle(add) -> remove.

init_boss(Source, Food, Operation) ->
    random:seed(erlang:now()),
    boss(Source, Food, Operation).


boss(_Source, [], _Operation) ->
    done;
boss(Source, [Food|Foods], Operation) ->
    timer:sleep(1000),
    timer:send_after(1000 + random:uniform(1000), Source, {Operation, Food}),
    boss(Source, Foods, toogle(Operation)).
```

`timer:send_after(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`

Evaluates **`Pid ! Message`** after Time amount of time has elapsed. Returns **`{ok, TRef}`**, or **`{error, Reason}`** **immediately.**




```
test(N, M) ->
    io:format("~w eaters, each eating ~w times~n", [N, M]),
    BadFood = ["Pizza", "Burger"],

    Source = spawn(?MODULE, init_food, [BadFood]),

    lists:map(fun(_) -> spawn(?MODULE, init_eater, [Source, M]) end, lists:seq(1, N)),

    BossFood = ["Pizza", "Banana", "Burger", "Pinaple"],
    spawn(?MODULE, init_boss, [Source, BossFood, remove]),

    all_spawned.
```

The **?MODULE** macro is often used together with spawn. **?MODULE** is replaced with the current module name by the compiler.

```
[spawn(?MODULE, init_eater, [Source, M]) || _ <- lists:seq(1, N)],
```

This is equivalent to using **list:map** when spawning the eater processes.

Test run

```
59> food:test(3,5).
3 eaters, each eating 5 times
all_spawned
<0.445.0> Pizza
<0.447.0> Pizza
<0.446.0> Burger
<0.444.0> Bad news, no more Pizza :(
<0.445.0> Burger
<0.446.0> Burger
<0.447.0> Burger
<0.444.0> God news, Banana now availalbe!
<0.447.0> Burger
<0.446.0> Burger
<0.445.0> Burger
<0.444.0> Bad news, no more Burger :(
<0.447.0> Banana
<0.444.0> God news, Pinaple now availalbe!
<0.446.0> Banana
<0.445.0> Pinaple
<0.447.0> Pinaple
<0.447.0> done
<0.446.0> Banana
<0.446.0> done
<0.445.0> Banana
<0.445.0> done
60>
```