Peer to Peer Computing (1DT047) summer 2009
Client Server Programming with Erlang
Wednesday June 17
karl.marklund@it.uu.se
A computer network (for example the Internet)
A client sends a request to a server

Client

Server

What time is it?

Current time: 13:17
A web browser is a HTML client sending requests to a web server

http://www.it.uu.se

<html>....</html>
A simple example:

"Message from client"

"Got it!"
The **server** uses a TCP socket to receive a message from the client and send a response.
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 {
            sock = new java.net.Socket(serverIPname, serverPort); // create socket and connect
            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");

            pw.println("Message from the client"); // send msg to the server

            System.out.println("Sent message to server");

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

            pw.close(); // close everything
            br.close();
            sock.close();
        }
        catch (Throwable e) {
            System.out.println("Error " + e.getMessage());
            e.printStackTrace();
        }
    }
}
server() ->
    io:format("waiting for client to connect~n"),
    receive
        Msg ->
            io:format("Message from the client > ~w~n", [Msg]),
            client ! "Got it!"
    end.

client() ->
    server ! "Message from the client",
    receive
        Msg -> io:format("Response from the server > ~s~n", [Msg])
    end.

We focus on the send and receive operations.

A more dense notation

Sending a message using the send operator!
A Erlang implementation

Sending and receiving messages over a network is very easy in Erlang
– it’s part of the language.
A **Erlang** implementation

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

start() ->
  register(server, self()),
  io:format("waiting for client to connect\n"),
  receive
    {Client, Msg} ->
      io:format("Message from the client > ~w\n", [Msg]),
      Client ! "Got it!"
  end.

-module(client).
-export([start/1]).

start(ServerNode) ->
  {server, ServerNode} ! {self(), "Message from the client"},
  receive
    Msg -> io:format("Response from the server > ~s\n", [Msg])
  end.
```

Sending and receiving messages over a network is very easy in Erlang

– it’s part of the language.

Yeah! The Erlang version is clearly more dense compared to the Java version – tell us more about Erlang.
In C-like languages such as C, C++ and Java we often use for-loops to iterate.

In Erlang we use functions and recursive function calls.

Erlang is a functional language...
Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and *avoids state and mutable data*.

Functional programming *emphasizes the application of functions*, in contrast to the imperative programming style, which emphasizes changes in state.
The factorial function implemented in Erlang

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

if 0, then return 1

recursively calculate (N-1)!

the period . meaning 'endif' or 'function end'

the semicolon ; meaning 'else'

Hmm... Erlang looks just like mathematical formulas.
Erlang modules

Erlang modules gives functions a namespace to live in.

File: example.erl

-module(example).
-export([fact/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).

Functions that we want to call from outside the module must be exported.

It is not *tail recursive* because the recursive call fac(N-1) is not the whole result - it has to be multiplied by N.
To write recursive functions efficiently in Erlang they must be tail recursive. A function is tail recursive (roughly speaking) if for any recursive call the answer returned by the recursive call is the whole answer.

Here is a tail recursive version. It uses a two-argument version of factorial. The second argument of fact/2 accumulates the result

```erlang
-module(example).
-export([fact/1]).

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

This is tail recursive, since the result of the recursive call fac(N-1,N*A) is the whole result.

This works by accumulating the result in the second parameter. Tail recursive functions are important since they can be compiled to run very efficiently. This is particularly important when writing functions which will be used as processes in Erlang.
**Tail recursion** (or tail-end recursion) is a special case of recursion in which the last operation of the function, the tail call, is a recursive call.

Such recursions *can be easily transformed to iterations*. Replacing recursion with iteration, manually or automatically, can drastically decrease the amount of stack space used and *improve efficiency*.

This technique is commonly used with functional programming languages and is *done automatically by the Erlang compiler*.

So there is not a valid argument to say that imperative languages (C, C++, Java, ...) implements iteration (for loops) more efficiently than functional languages (Erlang, Haskel, ML, ...) as long as tail recursion optimization is used.
The Erlang shell

Start the Erlang shell:

```
$ erl
Eshell V5.5.5 (abort with ^G)
1>
```
The Erlang shell

Start the Erlang shell

$ erl

Eshell V5.5.5 (abort with ^G)

1> c(example).

Compile the example.erl module

Must end every command (expression) with a dot.
The Erlang shell

Start the Erlang shell

The Erlang shell prompt

Result from compilation

$ erl
Eshell V5.5.5 (abort with ^G)
1> c(example).
{ok,example}
2> Compile the example.erl module

Must end every command (expression) with a dot.
The Erlang shell

Start the Erlang shell

Eshell V5.5.5 (abort with ^G)

The Erlang shell prompt

Compile the example.erl module

Result from compilation

Must end every command (expression) with a dot.

Evaluate the fact function in the example module with argument 0.
The Erlang shell

Start the Erlang shell

The Erlang shell prompt

Result from compilation

Erlang uses arbitrary-sized integers

Evaluate the fact function in the example module with argument 0.

Must end every command (expression) with a dot.

Compile the example.erl module

$ erl
Eshell V5.5.5 (abort with ^G)
1> c(example).
{ok,example}
2> example:fact(0).
1
3> example:fact(5).
120
4> example:fact(10).
3628800
5> example:fact(25).
15511210043330985984000000
Erlang Variables Are Not-So-Variable

$ erl

Eshell V5.5.5 (abort with ^G)

1> X = 1.
1
2> Assign integer value 1 to variable X
Erlang Variables Are Not-So-Variable

Assign integer value 1 to variable X

"Assign" new value 127 to variable X
Erlang Variables Are Not-So-Variable

$ erl

Eshell V5.5.5 (abort with ^G)

1> X = 1.
1
2> X = 127.
** exception error: no match of right hand side value 127

Assign integer value 1 to variable X

”Assign” new value 127 to variable X

Variable names start with a capital letter.

Cannot re-assign values to variables.

Variables are immutable: Erlang's notion of variables is closer to that of algebra than it is something like C or Java.
Atoms

An atom is a constant symbol. Atom names are in lower case.

```
1> fruit.
fruit
2>
```
Atoms

1> fruit.
fruit
2> vegetable = 3.

An atom is a constant symbol. Atom names are in lower case.
Atoms

1> fruit.
fruit
2> vegetable = 3.
** exception error: no match of right hand side value 3

An atom is a constant symbol. Atom names are in lower case.

Atoms cannot be assigned a value.
Atoms

1> fruit.
fruit
2> vegetable = 3.
** exception error: no match of right hand side value 3
3> vegetable =:= fruit.
false
4> Object = fruit.
fruit
5> Object =:= fruit.
true

An atom is a constant symbol. Atom names are in lower case.

Atoms cannot be assigned a value.

You can do equality comparisons on atoms using the is-identical-to operator =:=
A tuple is a set of fixed size.
Tuples

1> Point = {1,4,3}.
{1,4,3}
2> {X, Y, Z} = Point.
{1,4,3}
3> X.
1
4> Y.
4
5> Z.
3
6>
Tuples

A tuple is a set of fixed size.

Can use the match operator = to bind variables to tuple elements.

Tuples can have elements of different types.
The following are all examples of lists:

<table>
<thead>
<tr>
<th></th>
<th>L1 = [1,2,3,4].</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>L2 = [&quot;kittens&quot;,trees,&quot;bears&quot;,1,true].</td>
</tr>
<tr>
<td>3</td>
<td>L3 = [[{apple,banana},{cat,dog}],{tomato,avocado}].</td>
</tr>
</tbody>
</table>

We can separate the head from the tail of a list by using the | operator in a pattern:

|   | [H1|T1] = L1. |
|---|--------------|
| 5 | H1. 1       |
| 6 | T1. [2,3,4] |

Notice that the original list has not changed. We can use the same | operator to efficiently construct a new list by appending a value to the head of an existing list:

|   | L4 = [2|L2]. |
|---|--------------|
| 9 | L2. ["kittens",trees,"bears",1,true] |
Spawn a process which kicks off the loop() function
This form of spawn() requires the loop() function to be exported.

Start the loop again

Send an atom to the loop() process

Send a tuple to the loop() process

Spawn returns the process id (PID) of the new process
If statement

test_if(A, B) ->
  if
    A == 5 ->
      io:format("A = 5~n", []),
      a_equals_5;
    B == 6 ->
      io:format("B = 6~n", []),
      b_equals_6;
    A == 2, B == 3 -> %i.e. A equals 2 and B equals 3
      io:format("A == 2, B == 3~n", []),
      a_equals_2_b_equals_3;
    A == 1 ; B == 7 -> %i.e. A equals 1 or B equals 7
      io:format("A == 1 ; B == 7~n", []),
      a_equals_1_or_b_equals_7
  end.

Case statement

is_valid_signal(Signal) ->
  case Signal of
    {signal, _What, _From, _To} ->
      true;
    {signal, _What, _To} ->
      true;
    _Else ->
      false
  end.
Guards

Legal guards in Erlang are boolean functions placed after the key word, "when" and before the arrow, "->".

We can use a guard in a function definition

```
-module(guardian).
-export([export_all/1]).

the_answer_is(N) when N == 42 -> true;
the_answer_is(N) -> false.
```

```
1> c(guardian).
ok
2> guardian:the_answer_is(42).
true
3> guardian:the_answer_is(21).
false
```
Guards

receive expression

receive
  \{answer, N\} when N == 42 -> true;
  \{answer, N\} -> false
end.

if expression

if
  N == 42 -> true;
  true -> false
end.

case expression

case L of
  \{answer, N\} when N == 42 -> true;
  _ -> false
end.
For a good introduction to the Erlang programming language, see this excellent tutorial from where some of the examples in these slides are borrowed:


Also see this introduction to Erlang:

http://www.erlang.org/download/getting_started-5.4.pdf
A simple server keeps track on the number of client request.

A special owner process monitors the state of the server and can decide to reset the server hit counter.