The previous lecture left off talking about pattern matching in Erlang. As a quick addendum to this topic, it is important to note that, if properly used, the patterns in Erlang are non-overlapping.

The meat of the first half of the lecture focused on studying some examples of concurrency in Erlang, beginning with a naïve counter program with several inherent issues (such as a lack of direct access to the counter variable and non-implicit messaging). The code for this “naïve counter” is reprinted in Fig. 1 (below), with a refinement available in Fig. 2 (found on the following page).

```
-module(counter).
-export([[start/0,loop/1]]).
start() ->
  spawn(counter, loop, [0]).
loop(Val) ->
  receive
    increment -> loop(Val + 1)
  end.
```

Fig 1. “Naive counter” source

Clearly, the implementation found in Fig. 1 is not satisfactory. Although superficially the goal of our counter was fulfilled (to make an incrementing counter value), several needed features are missing. We can implement a better example by having a more robust message-passing implementation and an actual way to end the program. We can thus imagine a set of function-argument pairs called start(), increment(Counter), value(Counter), stop(Counter) and loop(Val), all with rather self-explanatory purposes and return values. We can also take advantage of this “refined counter” to demonstrate some of the intricacies of Erlang’s pattern matching system.
-module(counter).
-export([start/0, loop/1, increment/1, value/1, stop/1]).

%% First the interface functions.
start() ->
    spawn(counter, loop, [0]).

increment(Counter) -> Counter ! Increment.

value(Counter) ->
    Counter ! {self(), value},
    receive
        {Counter, Value} -> Value
    end.

stop(Counter) -> Counter ! Stop.

loop(Val) ->
    receive
        increment -> loop(Val + 1);
        {From, value} -> From ! {self(), Val}, loop(Val);
        stop -> true;
        Other -> loop(Val)
    end.

Fig 2. “Refined counter” source

increment(Counter) and stop(Counter) are fairly straightforward to understand. When they are called with any particular Counter as an argument, they pass an Increment or a Stop message to that particular Counter, respectively. loop(Val) then either increments the current value by one and continues looping (for an Increment signal), or simply returns true when a Stop signal is received.

value(Counter) is a bit less straightforward – when called, it sends a message consisting of a pair containing itself and a “value” name. Why is this pair necessary? Why can we not take the same approach as with the other two functions and simply pass a Value which we then receive in loop? The answer lies in Erlang's pattern matching, which, if invoked in this way, would overlap the received Value with (for example) Stop. We thus utilize a pair, despite the fact that we never use the second element of the pair in the loop call.

We can see now that we have a more robust counter program. We can now use this knowledge base to analyze a concurrent “ping/pong” program which spawns two processes and has them communicate with each other via message-passing until a certain condition is met. Consider Fig. 3 on the following page.
-module(M).
-export([start/0, ping/1, pong/0]).

ping(0) ->
    pong ! finished,
    io:format("ping finished-n", []).
ping(N) ->
    pong ! {ping, self()},
    receive pong ->
        io:format("Ping received pong-n", [])
    end,
    ping(N - 1).

pong() ->
    receive
        finished -> io:format("Pong finished-n", []).
        {ping, Ping_PID} ->
            io:format("Pong received ping-n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    register(pong, spawn(M, pong, [])),
    spawn(M, ping, [3]).

Fig 3. Ping / pong concurrency.

We can begin our analysis at start(), which registers the name “pong” and binds it to the act of spawning a pong process with no arguments inside module M. This ties pong’s process ID to an accessible name, so that pong can be available on the top level environment. start() then spawns the process ping with a preset arbitrary end condition (in this case, 3).

By looking at ping’s pattern matching, we can see the following course of events unfurl. As ping has received a non-zero argument, we look at ping(N), which dictates we pass a message to pong consisting of a pair – the name ping, and a self() call, which gives pong access to that particular ping’s process ID. We then wait until pong replies with a message, at which point we display “Ping received pong” and iterate once more with our end condition decremented by one. When the end condition reaches zero, we send a finished message to pong and display a relevant string to the screen.

What, then, does pong do? It simply waits for messages to be passed to it in one of two forms. If it received finished, then it displays a string and returns control to ping, which eventually terminates as well. If it receives a pair from ping, however, it displays a string to the screen, passes a pong message to the received Ping_PID and loops.
It is interesting to note that the preceding example can be readily converted to work in a distributed environment. This generalization would require some code changes, but none of them would change the control flow of the program, merely some of the top-level implementation details. The distributed source code can be found in Fig. 4 below.

```
-module(M).
-export([start/0, ping/1, pong/0]).

ping(0,Pong_node) ->
  {pong, Pong_Node} ! finished,
  io:format("ping finished-n", [],);
ping(N) ->
  {pong, Pong_Node} ! {ping, self()},
  receive pong ->
    io:format("Pong received pong-n", [],)
  end,
  ping(N - 1, Pong_Node).
pong() ->
  receive
    finished ->
      io:format("Pong finished-n", [],);
    {ping, Ping_PID} ->
      io:format("Pong received ping-n", [],),
      Ping_PID ! pong,
      pong()
  end.

start_pong() ->
  register(pong, spawn(M, pong, [])),
  start_ping(Pong_Node) ->
    spawn(M, ping, [3, Pong_Node]).
```

On one host:
erl -sname ping
On another:
erl -sname pong

On one node:
M:start_pong().
On another:
M:start_ping(pong@<host>).

```
Fig 4. Distributed ping / pong concurrency.
```

The lecture continued with a brief discussion of **monitoring**. A code snippet follows in Fig. 5, but it is sufficient to say that `spawn_link` can be bound to a variable and used to monitor your program, keeping track of things such as exceptions and pattern-match errors.

```
... process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
receive
  {'EXIT', Pid, Why} ->
  ... end
```

```
Fig 5. Monitoring snippet
```
We then considered a server/client setup, which involved the use of remote procedure calls. Consider the following basic server/client setup found in Fig. 6:

```erlang
server(Fun, Data) ->
    receive
    {new_fun, Fun1} ->
        server(Fun1, Data);
    {rpc, From, ReplyAs, Q} ->
        {Reply, Data1} = Fun(Q, Data),
        From ! {ReplyAs, Reply},
        server(Fun, Data1)
    end.

rpc(A, B) ->
    Tag = new_ref(),
    A ! {rpc, self(), Tag, B},
    receive
    {Tag, Val} -> Val
    end.
```

Fig 6. Server/client setup.

The remote procedure call (or RPC) is found in the aptly-named rpc(A, B) function, which does the following:

1. `Tag = new_ref()`, Creates a Tag bound to a new_ref(), which allows multiple RPCs in the same process, and allows differentiation of RPC calls.
2. `A ! {rpc, self(), Tag, B}`, Sends an RPC message with all of the relevant info to the server A.
3. `receive ... end`. Waits for the server to reply with the same Tag (meaning the appropriate RPC was answered) and reads data accordingly.

We can see that the server itself waits on messages. One possibility is that the server receives a message telling it to update its function, containing a new_fun message and the relevant new Fun1 to perform. Alternatively, the server could receive the message that the RPC would send, which would cause the server to perform some Fun on the Data, which would subsequently be stored in {Reply, Data1}. The process ID of the RPC (which was passed to the server in the RPC using self()) would then be sent a message containing a reply by the server, and the server would resume running.
The lecture was concluded with a discussion of concurrency patterns found in Erlang. There are many different basic concurrency patterns which are easily created and understood in Erlang – indeed, we have already seen that we can define four very important concurrency patterns by only using basic sends and receives. An overview of these patterns is found in Fig. 7.

**Unicast**

A ! B

**RPC call**

Call (RPC)

A ! {self(), B},

receive

{A, Reply} -> Reply

end

**Event Handling**

receive

A -> A

end

**Callback**

receive

{From, A} ->

From ! F(A)

end

Fig 7. Basic send/receive concurrency patterns in Erlang

This concluded our overview of Erlang, and will hopefully serve as the basis for a fruitful understanding of both Erlang as a language, and its relation to and importance within concurrent and parallel programming.