

## Priority Messaging made Easy A Selective Generic Server

ACM SIGPLAN Erlang Workshop  
Freiburg, Germany, October 4<sup>th</sup> 2007

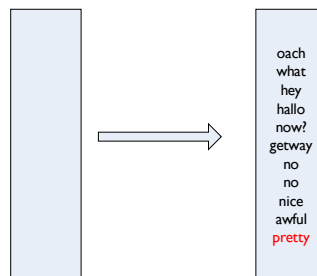


Erlang Training and Consulting Ltd  
[www.erlang-consulting.com](http://www.erlang-consulting.com)

Jan Henry Nyström  
[jan@erlang-consulting.com](mailto:jan@erlang-consulting.com)

## Priorities Take 1, Silence, Action

```
prio() ->  
  receive pretty -> do_pretty()  
  after 0 -> receive X -> whatever(X)  
end.
```



## Priorities Nearly there

```
prio() ->  
  receive pretty -> do_pretty()  
  after 0 -> receive X -> prio1(X)  
end.
```

```
prio1(pretty) -> do_pretty();  
prio1(X) ->  
  receive pretty -> do_pretty()  
  after 0 -> do_x(X)  
end.
```

## That's a wrap

```
prio() ->  
  case get(peek) of  
  undefined ->  
    receive pretty -> do_pretty()  
    after 0 -> receive X -> prio1(X)  
  end;  
  X ->  
    receive pretty -> do_pretty()  
    after 0 -> put(peek, undefined), do_x(X)  
  end  
end.
```

```
prio1(pretty) -> do_pretty();  
prio1(X) ->  
  receive pretty -> put(peek, X), do_pretty()  
  after 0 -> do_x(X)  
end.
```

## Act 1, Scene 2, Several Priorities...

```
prio! ->
case get(peek) of
  undefined ->
    receive pretty -> do_pretty()
    after 0 -> receive nice -> do_nice()
              after receive X -> prio!(X)
    end
  end;
X ->
  receive pretty -> do_pretty()
  after 0 -> put(peek, undefined),
            case X of
              nice -> put(peek, undefined),
                    do_nice();
              _ -> do_x(X)
            end
  end
end.

prio!(pretty) -> do_pretty();
prio!(nice) -> receive pretty -> put(peek, nice),
              do_pretty()
              after 0 -> do_nice() end;
prio!(X) ->
  receive pretty -> put(peek, X), do_pretty()
  after 0 -> receive nice -> put(peek, X), do_nice()
  after 0 -> do_x(X) end
end.
```

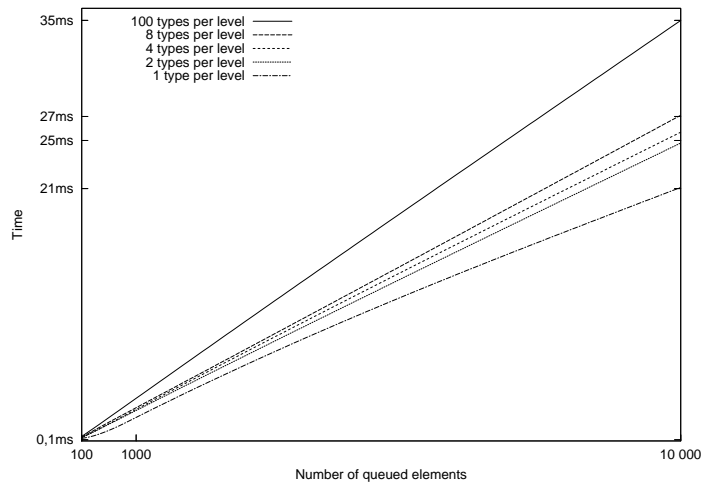
## Warning words from the Guru

***“Using large mailboxes with priority receive is rather inefficient, so if you’re going to use this technique, make sure your mailboxes are not too large.” Joe Armstrong***

In fact, in our case the complexity is:

- (the number of priority levels) times
- (the number of clauses in each priority level) times
- (the number of messages in the mailbox) times
- (the number of messages in the mailbox).

## The Price We Pay for Priorities



## The Golden Rules of Priority Messaging

1. Only use priority message reception if you really, really need it.
2. Limit the number of messages on the sending side.
3. Only the necessary priority levels.

## A new Behaviour

```
-module(test_select_server).
-behaviour(gen_select_server).
-export([start_link/0, call/1, ...]).

-selection([{{call, {first, '$1', '$1'}},
            {all, nepp}},
           {cast, {second, '$1', '$1'}}]).
start_link() -> gen_select_server:start_link({local,?MODULE},?MODULE,no_args,[]).
...
init(no_args) -> {ok, none}.

handle_call(Msg, _, none) ->
io:format("Call:[~p]~n", [Msg]),
{reply, ok, none}.
...
```

## Compiler Changes

```
compile_options([{{attribute,_L,behavior,C}|Fs]) ->
  behaviour_option(C, Fs);
compile_options([{{attribute,_L,behaviour,C}|Fs]) ->
  behaviour_option(C, Fs);
compile_options([{{attribute,_L,compile,C}|Fs]) when is_list(C) ->
  C ++ compile_options(Fs);
compile_options([{{attribute,_L,compile,C}|Fs]) ->
  [C|compile_options(Fs)];
compile_options([_F|Fs]) -> compile_options(Fs);
compile_options([]) -> [].
```

```
behaviour_option(C, Fs) ->
case catch C:behaviour_info(parse_transform) of
true -> [{{parse_transform, C}|compile_options(Fs)];
_ -> compile_options(Fs)
end.
```

## The Proof is in the Eating of the Pudding

A small testcase:

```
cast(duu),
cast(nepp),
call(third), cast(third), info(third),
call({first, 1, 2}), cast({first, 1, 2}), info({first, 1, 2}),
cast({second, a, a}), call({second, a, a}),info({second, a, a}),
info({first, o, o}), cast({first, o, o}),call({first, o, o}).
```

## Tests are running

**No Priorities:**

```
2> test_select_server:test().
Cast:[duu]
Cast:[nepp]
Cast:[third]
Info:[third]
Cast:[{first, 1, two}]
Info:[{first, 1, two}]
Cast:[{second, a, a}]
Info:[{second, a, a}]
Info:[{first, o, o}]
Cast:[{first, o, o}]
Call:[third]
Call:[{first, 1, two}]
Call:[{second, a, a}]
Call:[{first, o, o}]
```

**Priorities:**

```
2> test_select_server:test().
Cast:[nepp]
Call:[{first, o, o}]
Cast:[{second, a, a}]
Cast:[duu]
Cast:[third]
Info:[third]
Cast:[{first, 1, two}]
Info:[{first, 1, two}]
Info:[{second, a, a}]
Info:[{first, o, o}]
Cast:[{first, o, o}]
Call:[third]
Call:[{first, 1, two}]
Call:[{second, a, a}]
```

## Did anyone say EEP

The power of behaviours would be greatly improved if they could invoke parse transforms implicitly by adding information in the `behaviour_info/1` call back function in the implementing thus enabling compile time changes.

### Rationale

The ability would make the creation of new behaviours easier by allowing skeleton behaviours to be added that, together with a few compiler directives generate new behaviours extending the skeleton (or indeed fully fledged stand alone) behaviours. Thus new behaviours could be constructed with a minimum of coding and sometimes completely bypassing the difficult parts of writing behaviours. For an example see [1].

### Changes Necessary

The only part of OTP that needs to be changed is the compiler and linter. In compiler the part of `compile.erl` that reads:

## Concluding words

- **Priority messaging is hard.**
- **Writing new behaviours is easy.**
- **Writing new behaviours should be made even easier.**