

Communication Patterns

CS511

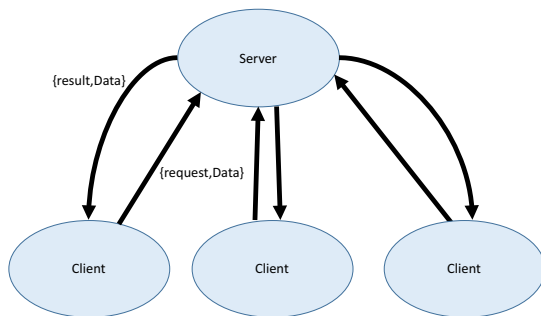
Client Server Architecture

A Generic Server

Concurrency Patterns

Client-Server Architecture

- ▶ Common asynchronous communication pattern
- ▶ For example: a web server handles requests for web pages from clients (web browsers)



Example: Factorial Server

```
1 -module(mserver).
2 -export([start/0,compute_factorial/2]).
3 -import(fact,[fact/1]).
4
5 loop(Count) ->
6     receive
7         {get_count, From, Ref} ->
8             From ! {result, Ref, Count},
9             loop(Count);
10
11         {factorial, From, Ref, N} ->
12             Result = fact(N),
13             From ! {result, Ref, Result},
14             loop(Count+1);
15
16     stop -> true
17 end.
18
19 % starting server with initial state 0
20 start() -> spawn(fun() -> loop(0) end).
```

Note how the server state is a parameter of loop

Example: Factorial Server

Client

```
1 compute_factorial(Pid, N) ->
2   Ref = make_ref(),
3   Pid ! {factorial, self(), Ref, N},
4   receive
5     {result, Ref, Result} ->
6       Result
7   end.
```

Test

```
1 > c(mserver).
2 {ok,mserver}
3 > P=mserver:start().
4 <0.40.0>
5 > mserver:compute_factorial(P,10).
6 3628800
```

Example: Factorial Server

What if the server crashes or stops?

```
1 > P ! stop
2 > mserver:compute_factorial(P,10).
3 ...no response...
```

- ▶ Why do we get no response?
- ▶ Can you modify the code so that we receive a `timeout`?

Registered Processes – Recap

- ▶ As seen in class, Erlang has a method for publishing a process identifier

- ▶ Any other process can communicate with it

- ▶ BIF `register`

```
1  % starting server with initial state 0
2  start() ->
3      Pid = spawn(fun() -> loop(0) end),
4      register(server, Pid).
```

- ▶ Unregister with `unregister(name)`
- ▶ Registration lookup `whereis(name)`

Registered Processes – Recap

The atom server can be used instead of a concrete process ID

```
1 > mserver2:start().
2 true
3 > mserver2:compute_factorial(server,10).
4 3628800
5 > server ! stop.
6 stop
7 > mserver2:compute_factorial(server,10).
8 ** exception error: bad argument
9     in function mserver2:compute_factorial/2 (mserver2.erl, line
10 > mserver2:start().
11 true
12 > mserver2:compute_factorial(server,10).
13 3628800
```


Distributed Environments

- ▶ Message passing abstractions extend easily for distributed environments
- ▶ Erlang nodes
 - ▶ An instance of an Erlang runtime system
 - ▶ Nodes can easily communicate with each other
 - ▶ Creating a node

```
erl -name 'nodeS@127.0.0.1' -setcookie lecture
```

- ▶ The cookie provides security (not everyone can connect)
- ▶ The name reflects the node's IP address

Distributed Environments

► Creating two nodes (for simplicity on the same machine)

```
1 erl -name 'nodeS@127.0.0.1' -setcookie lecture
2 erl -name 'nodeC@127.0.0.1' -setcookie lecture
```

► Connecting nodes

► From nodeC@127.0.0.1

```
1 (nodeC@127.0.0.1)> net_adm:ping('nodeS@127.0.0.1').
2 pong
3 (nodeC@127.0.0.1)> nodes().
4 ['nodeS@127.0.0.1']
```

Distributed Factorial Server – Running Your Code

Send the compiled version of your code to the connected nodes

```
1 (nodeC@127.0.0.1)> nl(fact).  
2 abcast  
3 (nodeC@127.0.0.1)> nl(mserver2).  
4 abcast
```

The server gets started on the nodeS node

```
1 (nodeS@127.0.0.1)> mserver2:start().  
2 true
```

The client communicates with the server

```
(nodeC@127.0.0.1)> mserver2:compute_factorial({server,  
                                              'nodeS@127.0.0.1'}, 10).  
3628800
```

- ▶ Use of `{registered_name, node@IP}` instead of the pid or only the registered name
- ▶ Code has not been changed for running in a distributed setting!

Client Server Architecture

A Generic Server

Concurrency Patterns

A Generic Server

- ▶ The code for a generic server takes care of the communication, faults, and upgrades
- ▶ Programmers then only focus on writing the engine (i.e. what the server does)
- ▶ No communication primitives are required in the engine

A Generic Server

The code must expose the following features:

- ▶ Correct
 - ▶ It implements a proper server/client request/reply interaction
- ▶ Parametrized
 - ▶ It is parametric on the engine
- ▶ Robust
 - ▶ It does not crash if the engine goes wrong
- ▶ Upgradable
 - ▶ It allows to upgrade the engine of the server without shutting it down

A Generic Server

```
1 loop(State, F) ->
2   receive
3     {update, From, Ref, NewF} ->
4       From ! {ok, Ref},
5       loop(State, NewF) ;
6
7     {request, From, Ref, Data} ->
8       {R, NS} = F(State, Data),
9       From ! {result, Ref, R},
10      loop(NS, F);
11
12   stop -> true
13
14 end.
```

How can the server go wrong when evaluating $F(\text{State}, \text{Data})$?

Exceptions – The evaluation of expressions can fail

► Arithmetic error

```
1 > 1/0.  
2 ** exception error: bad argument in an arithmetic expression  
3    in operator  '/' /2  
4        called as 1 / 0
```

► Bad pattern matching

```
1 [] = [1].  
2 ** exception error: no match of right hand side value [1]
```

► Undefined functions

```
1 net_adm:ping(1,2).  
2 ** exception error: undefined function net_adm:ping/2
```


Exceptions

```
1 > catch(1/0).
2 {'EXIT',{badarith,[{erlang,'/',[1,0]},
3                     {erl_eval,do_apply,5},
4                     {erl_eval,expr,5},
5                     {shell,exprs,7},
6                     {shell,eval_exprs,7},
7                     {shell,eval_loop,3}]}}}
8 > catch([] = [1]).
9 {'EXIT',{badmatch,[1]},{erl_eval,expr,3}}}]
10 > catch(net_adm:ping(1,2)).
11 {'EXIT',{undef,[{net_adm,ping,[1,2]},
12                 {erl_eval,do_apply,5},
13                 {erl_eval,expr,5},
14                 {shell,exprs,7},
15                 {shell,eval_exprs,7},
16                 {shell,eval_loop,3}]}}}
17 >
```

Exceptions

```
1 loop(State, F) ->
2   receive
3     {update, From, Ref, NewF} ->
4       From ! {ok, Ref},
5       loop(State, NewF);
6
7     {request, From, Ref, Data} ->
8     ,,, case catch(F(State, Data)) of
9         {'EXIT', Reason} ->
10           From!{exit, Ref, Reason},
11           loop(State, F);
12         {R, NewState} ->
13           From!{result, Ref, R},
14           loop(NewState, F)
15         end;
16
17   stop -> true
18 end.
```

- It propagates the exception from the server to the client

Exceptions

```
1 loop(State, F) ->
2     receive
3         {update, From, Ref, NewF} ->
4             From ! {ok, Ref},
5             loop(State, NewF);
6
7         {request, From, Ref, Data} ->
8             case catch(F(State, Data)) of
9             , , {'EXIT', Reason} ->
10                , , From!{exit, Ref, Reason},
11                , , loop(State, F);
12                {R, NewState} ->
13                    From!{result, Ref, R},
14                    loop(NewState, F)
15                end;
16
17     stop -> true
18 end.
```

- It propagates the exception from the server to the client

Exceptions

```
1 loop(State, F) ->
2   receive
3     {update, From, Ref, NewF} ->
4       From ! {ok, Ref},
5       loop(State, NewF);
6
7     {request, From, Ref, Data} ->
8       case catch(F(State, Data)) of
9         {'EXIT', Reason} ->
10           From!{exit, Ref, Reason},
11           loop(State, F);
12       , {R, NewState} ->
13         From!{result, Ref, R},
14         loop(NewState, F)
15       , end;
16
17   stop -> true
18 end.
```

- It propagates the exception from the server to the client

Starting the Generic Server

```
1 start(Name, State, F) ->
2   Pid = spawn(fun() -> loop(State, F) end),
3   register(Name, Pid),
4   Pid.
```

Generic Client

► Requests

```
1 request(Pid, Data) ->
2   Ref = make_ref(),
3   Pid!{request, self(), Ref, Data},
4   receive
5     {result, Ref, Result} ->
6       Result;
7     {exit, Ref, Reason} ->
8       exit(Reason)
9   end.
```

Generic Client

► Upgrading the server's engine

```
1 update(Pid, Fun) ->
2   Ref = make_ref(),
3   Pid!{update, self(), Ref, Fun},
4   receive
5     {ok, Ref} ->
6       ok
7   end.
```

Factorial Server Revisited

```
1 -module(factServer).
2 -export([start/0,compute_factorial/1]).
3 -import(fact,[fact/1]).
4
5 engine(Count, {factorial,N}) ->
6     Result = math_examples:factorial(N),
7     {Result, Count+1} ;
8
9 engine(Count, get_count) ->
10    {Count, Count}.
11
12 start() ->
13     genserver:start(server, 0, fun engine/2).
14
15 compute_factorial(N) ->
16     genserver:request(server, {factorial, N}).
```

- Observe that there are no message passing primitives!

Factorial Server Revisited

```
1 4> factServer:start().  
2 <0.69.0>  
3 5> factServer:compute_factorial(23).  
4 25852016738884976640000
```

Client Server Architecture

A Generic Server

Concurrency Patterns

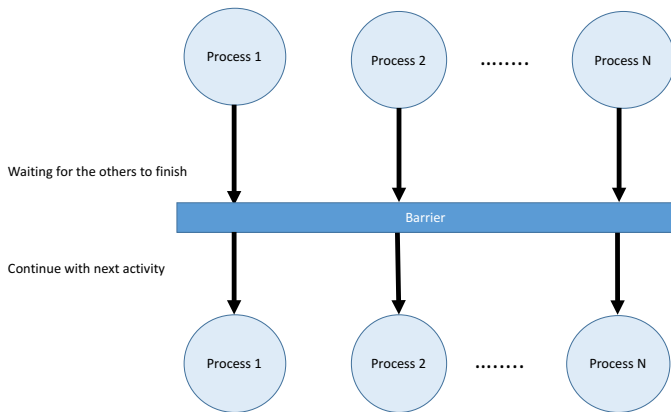
Concurrency Examples and Patterns Revisited

Revisiting the following using message passing:

- ▶ A semaphore (already seen last class)
- ▶ Barrier synchronisation
- ▶ Resource allocation
- ▶ Readers and writers

Barrier Synchronization Revisited

- ▶ N processes must wait for the slowest before continuing with the next activity
- ▶ Widely used in parallel programming



Barrier Synchronization Revisited

```
1 start(N) ->
2   Pid = spawn(fun() -> coordinator(N,N,[]) end),
3   register(coordinator, Pid).
4
5 coordinator(N,0,Ps) ->
6   [ From ! {ack, Ref} || {From, Ref} <- Ps ],
7   coordinator(N,N,[]) ;
8
9 coordinator(N,M,Ps) ->
10  receive
11    {reach, From, Ref} ->
12      coordinator(N,M-1, [ {From,Ref} | Ps])
13  end.
```

Barrier Synchronization Revisited

Using the barrier

```
1 reach_wait(Server) ->
2   Ref = make_ref(),
3   Server ! {reach, self(), Ref},
4   receive
5     {ack, Ref} -> true
6   end.
```

Resource Allocation

- ▶ A controller controls access to copies of some resources (of the same kind)
- ▶ Clients requiring multiple resources should not ask for resources one at a time
- ▶ Clients make requests to take or return any number of the resources
 - ▶ A request should only succeed if there are sufficiently many resources available
 - ▶ Otherwise the request must block

Resource Allocation

```
1 > c(ralloc).  
2 {ok, ralloc}  
3 > ralloc:start([1,1,1,1]).  
4 true  
5 > ralloc:request(3).  
6 [1,1,1]  
7 > ralloc:release([1]).  
8 ok  
9 > ralloc:request(2).  
10 [1,1]  
11 > ralloc:request(10).
```

In the last line, the process blocks

Resource Allocation

```
1 loop(Resources) ->
2   Available = length(Resources),
3   receive
4     {req, From, Ref, Number} when Number <= Available ->
5       From ! {res, Ref, lists:sublist(Resources, Number)},
6       loop(lists:sublist(Resources, Number+1, Available)) ;
7
8     {ret, List} -> loop(lists:append(Resources, List))
9   end.
10
11 % continues...
```

► Function `lists:sublist` returns a slice of a list; Examples

```
1 > lists:sublist([1,2,3,4], 2).
2 [1,2]
3 > lists:sublist([1,2,3,4], 2, 2).
4 [2,3]
5 > lists:sublist([1,2,3,4], 2, 5).
6 [2,3,4]
7 > lists:sublist([1,2,3,4], 5, 2).
8 []
```

Resource Allocation

```
1 start(Init) ->
2     Pid = spawn (fun () -> loop(Init) end),
3     register(rserver, Pid).
4
5
6 request(N) ->
7     Ref = make_ref(),
8     rserver ! {req, self(), Ref, N},
9     receive
10         {res, Ref, List} -> List
11     end.
12
13 release(List) ->
14     rserver ! {ret, List},
15     ok
```

Readers and Writers Revisited

- ▶ Two kinds of processes share access to a “database”
- ▶ Readers examine the contents
 - ▶ Multiple readers allowed concurrently
- ▶ Writers examine and modify data
 - ▶ A writer must have mutex
- ▶ Readers and writers in a few lines

Readers and Writers Revisited

```
1 loop(Rs, Ws) ->
2   receive
3     {start_read, From, Ref} when Ws == 0 ->
4       From ! {ok_to_read, Ref},
5       loop(Rs+1, Ws) ;
6
7     {start_write, From, Ref} when Ws == 0 and Rs == 0 ->
8       From ! {ok_to_write, Ref},
9       loop(Rs, Ws+1) ;
10
11   end_read -> loop(Rs-1, Ws) ;
12
13   end_write -> loop(Rs, Ws-1)
14 end.
```

Is it a fair solution?

Readers and Writers Revisited

```
1 loop(Rs, Ws) ->
2   receive
3     {start_read, From, Ref} when Ws == 0 ->
4       From ! {ok_to_read, Ref},
5       loop(Rs+1, Ws) ;
6
7     {start_write, From, Ref} when Ws == 0 and Rs == 0 ->
8       From ! {ok_to_write, Ref},
9       loop(Rs, Ws+1) ;
10
11   end_read -> loop(Rs-1, Ws) ;
12
13   end_write -> loop(Rs, Ws-1)
14 end.
```

Is it a fair solution? Unfair for writers

Fair Readers and Writers

```
1 loop() ->
2   receive
3     {start_read, From, Ref} ->
4       From ! {ok_to_read, Ref},
5       loop_read(1),
6       loop() ;
7
8     {start_write, From, Ref} ->
9       From ! {ok_to_write, Ref},
10      receive
11        end_write -> loop()
12      end
13  end.
```

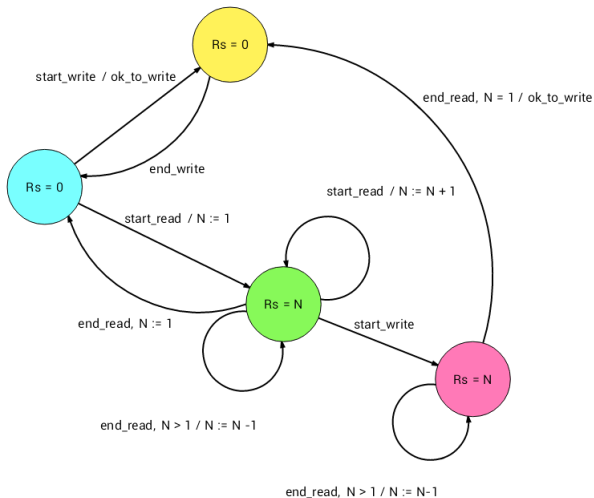
Fair Readers and Writers

```
1 loop_read(0) -> ok ;
2 loop_read(Rs) ->
3     receive
4         {start_read, From, Ref} ->
5             From ! {ok_to_read, Ref},
6             loop_read(Rs+1) ;
7
8     end_read -> loop_read(Rs-1) ;
9
10    {start_write, From, Ref} ->
11        [ receive end_read -> ok end
12          || _ <- lists:seq(1,Rs) ],
13        From ! {ok_to_write, Ref},
14        receive
15            end_write -> ok
16        end
17    end.
```

- ▶ At top-level `loop` relies on the fairness property of Erlang (i.e. the oldest message that matches any guard is processed)
- ▶ Function `loop_read` implements fairness
- ▶ Line `[receive end_read ->ok end || _ <- lists:seq(1,Rs)]` performs as many receive as the number `R`s

Fair Readers and Writers

- A FSM that describes its behavior



- Format of events:

`<received event>, <condition> / <triggered event>`