Communication Patterns

CS511

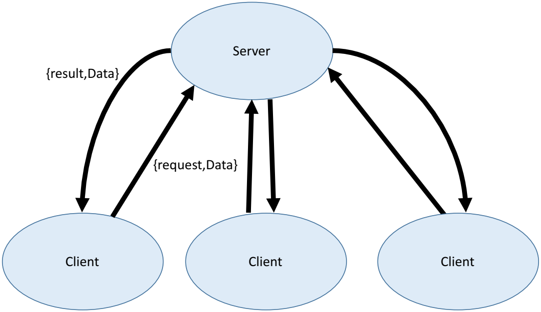
Client Server Architecture

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

```
-module(mserver).
-export([start/0,compute_factorial/2]).
-import(fact,[fact/1]).

loop(Count) ->
    receive
        {get_count, From, Ref} ->
                        From ! {result, Ref, Count},
                        loop(Count);

        {factorial, From, Ref, N} ->
                        Result = fact(N),
                        From ! {result, Ref, Result},
                        loop(Count+1);

        stop -> true
    end.

% starting server with initial state 0
start() -> spawn(fun() -> loop(0) end).
```

Note how the server state is a parameter of loop

Example: Factorial Server

Client

```
compute_factorial(Pid, N) ->
    Ref = make_ref(),
    Pid ! {factorial, self(), Ref, N},
    receive
        {result, Ref, Result} ->
                Result
    end.
```

Test

```
> c(mserver).
{ok,mserver}
> P=mserver:start().
<0.40.0>
> mserver:compute_factorial(P,10).
3628800
```

Example: Factorial Server

What if the server crashes or stops?

```
> P ! stop
> mserver:compute_factorial(P,10).
...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`

```
% starting server with initial state 0
start() ->
    Pid = spawn(fun() -> loop(0) end),
    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

```
> mserver2:start().
true
> mserver2:compute_factorial(server,10).
3628800
> server ! stop.
stop
> mserver2:compute_factorial(server,10).
** exception error: bad argument
     in function  mserver2:compute_factorial/2 (mserver2.erl, line
> mserver2:start().
true
> mserver2:compute_factorial(server,10).
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)

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

- Connecting nodes
  - From `nodeC@127.0.0.1`

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

Distributed Factorial Server – Running Your Code

Send the compiled version of your code to the connected nodes

```
(nodeC@127.0.0.1)> nl(fact).
abcast
(nodeC@127.0.0.1)> nl(mserver2).
abcast
```

The server gets started on the nodeS node

```
(nodeS@127.0.0.1)> mserver2:start().
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

```erlang
loop(State, F) ->
    receive
        {update, From, Ref, NewF} ->
                From ! {ok, Ref},
                loop(State, NewF) ;

        {request, From, Ref, Data} ->
                 {R, NS} = F(State, Data),
                 From ! {result, Ref, R},
                 loop(NS, F);

        stop -> true

    end.
```

How can the server go wrong when evaluating `F(State,Data)`?

Exceptions – The evaluation of expressions can fail

- Arithmetic error

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

- Bad pattern matching

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

- Undefined functions

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

Exceptions

```
> catch(1/0).
{'EXIT',{badarith,[{erlang,'/',[1,0]},
                   {erl_eval,do_apply,5},
                   {erl_eval,expr,5},
                   {shell,exprs,7},
                   {shell,eval_exprs,7},
                   {shell,eval_loop,3}]}}
> catch([] = [1]).
{'EXIT',{{badmatch,[1]},[{erl_eval,expr,3}]}}
> catch(net_adm:ping(1,2)).
{'EXIT',{undef,[{net_adm,ping,[1,2]},
                {erl_eval,do_apply,5},
                {erl_eval,expr,5},
                {shell,exprs,7},
                {shell,eval_exprs,7},
                {shell,eval_loop,3}]}}
>
```

Exceptions

```
loop(State, F) ->
    receive
        {update, From, Ref, NewF} ->
                  From ! {ok, Ref},
                  loop(State, NewF);

        {request, From, Ref, Data} ->
                   case catch(F(State, Data)) of
                       {'EXIT', Reason} ->
                              From!{exit, Ref, Reason},
                              loop(State, F);
                       {R, NewState} ->
                              From!{result, Ref, R},
                              loop(NewState, F)
                              end;

        stop -> true
end.
```

- It propagates the exception from the server to the client

Exceptions

```
loop(State, F) ->
    receive
        {update, From, Ref, NewF} ->
                 From ! {ok, Ref},
                 loop(State, NewF);

        {request, From, Ref, Data} ->
                  case catch(F(State, Data)) of
                      {'EXIT', Reason} ->
                             From!{exit, Ref, Reason},
                             loop(State, F);
                      {R, NewState} ->
                             From!{result, Ref, R},
                             loop(NewState, F)
                             end;

        stop -> true
end.
```

- It propagates the exception from the server to the client

Exceptions

```
loop(State, F) ->
    receive
        {update, From, Ref, NewF} ->
                 From ! {ok, Ref},
                 loop(State, NewF);

        {request, From, Ref, Data} ->
                  case catch(F(State, Data)) of
                      {'EXIT', Reason} ->
                             From!{exit, Ref, Reason},
                             loop(State, F);
                      {R, NewState} ->
                             From!{result, Ref, R},
                             loop(NewState, F)
                             end;

        stop -> true
end.
```

- It propagates the exception from the server to the client

Starting the Generic Server

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

Generic Client

- Requests

```
request(Pid, Data) ->
    Ref = make_ref(),
    Pid!{request, self(), Ref, Data},
    receive
        {result, Ref, Result} ->
            Result;
        {exit, Ref, Reason} ->
            exit(Reason)
    end.
```

Generic Client

- Upgrading the server's engine

```
update(Pid, Fun) ->
    Ref = make_ref(),
    Pid!{update, self(), Ref, Fun},
    receive
      {ok, Ref} ->
          ok
    end.
```

Factorial Server Revisited

```
-module(factServer).
-export([start/0,compute_factorial/1]).
-import(fact,[fact/1]).

engine(Count, {factorial,N}) ->
    Result = math_examples:factorial(N),
    {Result, Count+1} ;

engine(Count, get_count) ->
    {Count, Count}.

start() ->
    genserver:start(server, 0, fun engine/2).

compute_factorial(N) ->
    genserver:request(server, {factorial, N}).
```

- Observe that there are no message passing primitives!

Factorial Server Revisited

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

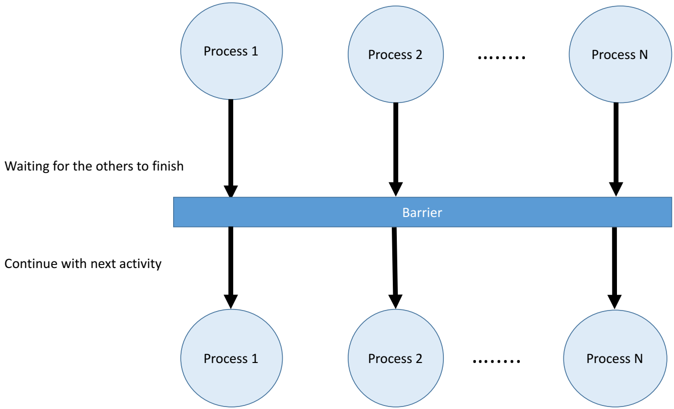
Concurrency Examples and Patterns Revisited

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

```
start(N) ->
    Pid = spawn(fun() -> coordinator(N,N,[]) end),
    register(coordinator, Pid).

coordinator(N,0,Ps) ->
    [ From ! {ack, Ref} || {From, Ref} <- Ps ],
    coordinator(N,N,[]) ;

coordinator(N,M,Ps) ->
    receive
        {reach, From, Ref} ->
                coordinator(N,M-1, [ {From,Ref} | Ps])
    end.
```

Barrier Synchronization Revisited

Using the barrier

```
reach_wait(Server) ->
    Ref = make_ref(),
    Server ! {reach, self(), Ref},
    receive
       {ack, Ref} -> true
    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

```
> c(ralloc).
{ok,ralloc}
> ralloc:start([1,1,1,1]).
true
> ralloc:request(3).
[1,1,1]
> ralloc:release([1]).
ok
> ralloc:request(2).
[1,1]
> ralloc:request(10).
```

In the last line, the process blocks

Resource Allocation

```
loop(Resources) ->
    Available = length(Resources),
    receive
       {req, From, Ref, Number} when Number =< Available ->
            From ! {res, Ref, lists:sublist(Resources, Number)},
            loop(lists:sublist(Resources, Number+1, Available)) ;

       {ret, List} -> loop(lists:append(Resources, List))
    end.

% continues...
```

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

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

Resource Allocation

```
start(Init) ->
    Pid = spawn (fun () -> loop(Init) end),
    register(rserver, Pid).


request(N) ->
    Ref = make_ref(),
    rserver ! {req, self(), Ref, N},
    receive
        {res, Ref, List} -> List
    end.

release(List) ->
    rserver ! {ret, List},
    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

```
loop(Rs, Ws) ->
    receive
       {start_read, From, Ref} when Ws =:= 0 ->
            From ! {ok_to_read, Ref},
            loop(Rs+1,Ws) ;

       {start_write, From, Ref} when Ws =:= 0 and Rs =:= 0 ->
            From ! {ok_to_write, Ref},
            loop(Rs, Ws+1) ;

       end_read  -> loop(Rs-1, Ws) ;

       end_write -> loop(Rs, Ws-1)
    end.
```

Is it a fair solution?

Readers and Writers Revisited

```
loop(Rs, Ws) ->
    receive
       {start_read, From, Ref} when Ws =:= 0 ->
            From ! {ok_to_read, Ref},
            loop(Rs+1,Ws) ;

       {start_write, From, Ref} when Ws =:= 0 and Rs =:= 0 ->
            From ! {ok_to_write, Ref},
            loop(Rs, Ws+1) ;

       end_read  -> loop(Rs-1, Ws) ;

       end_write -> loop(Rs, Ws-1)
    end.
```

Is it a fair solution? Unfair for writers

Fair Readers and Writers

```
loop() ->
    receive
       {start_read, From, Ref} ->
            From ! {ok_to_read, Ref},
            loop_read(1),
            loop() ;

       {start_write, From, Ref} ->
            From ! {ok_to_write, Ref},
            receive
                end_write -> loop()
            end
    end.
```

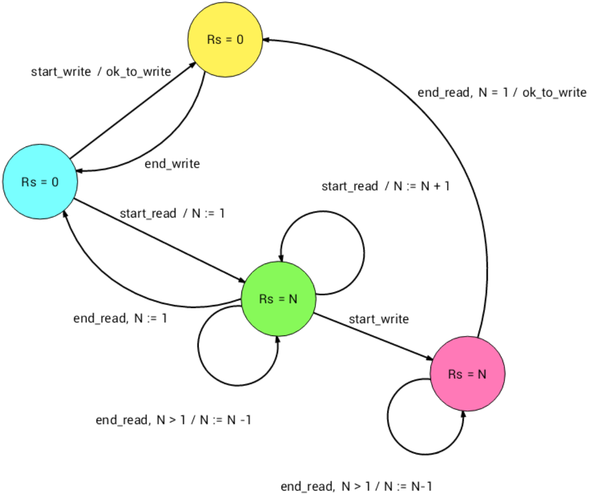
Fair Readers and Writers

```
loop_read(0) -> ok ;
loop_read(Rs) ->
    receive
       {start_read, From, Ref} ->
            From ! {ok_to_read, Ref},
            loop_read(Rs+1) ;

       end_read -> loop_read(Rs-1) ;

       {start_write, From, Ref} ->
            [ receive end_read -> ok end
              || _ <- lists:seq(1,Rs) ],
            From ! {ok_to_write, Ref},
            receive
                   end_write -> ok
            end
     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 Rs

Fair Readers and Writers

- A FSM that describes its behavior



- Format of events:

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