Message Passing Concurrency in Erlang

Joe Armstrong

Background

Observation B: Recently, I have been meeting a lot of Erlang people, and I sense clearly that they have this enviable ability to think intuitively about parallel programming. It corresponds somewhat to the way we "object heads" think intuitively about classes and objects - just in terms of processes.

Modeling Concurrency with Actors in Java - Lessons learned from Erjang

Kresten Krab Thorup, Hacker, CTO of Trifork


GREAT
IDEA

How do we think about parallel programs?

Using the wrong abstractions makes life artificially difficult

XLVIII x

XCIII

=

MMMMCDLXIV

The Big Idea is Messaging

From: Alan Kay <alank@wdi.disney.com>
Date: 1998-10-10 07:39:40 +0200
To: squeak@cs.uiuc.edu
Subject: Re: prototypes vs classes was: Re: Sun's HotSpot

Folks --

Just a gentle reminder that I took some pains at the last OOPSLA to try to remind everyone that Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.

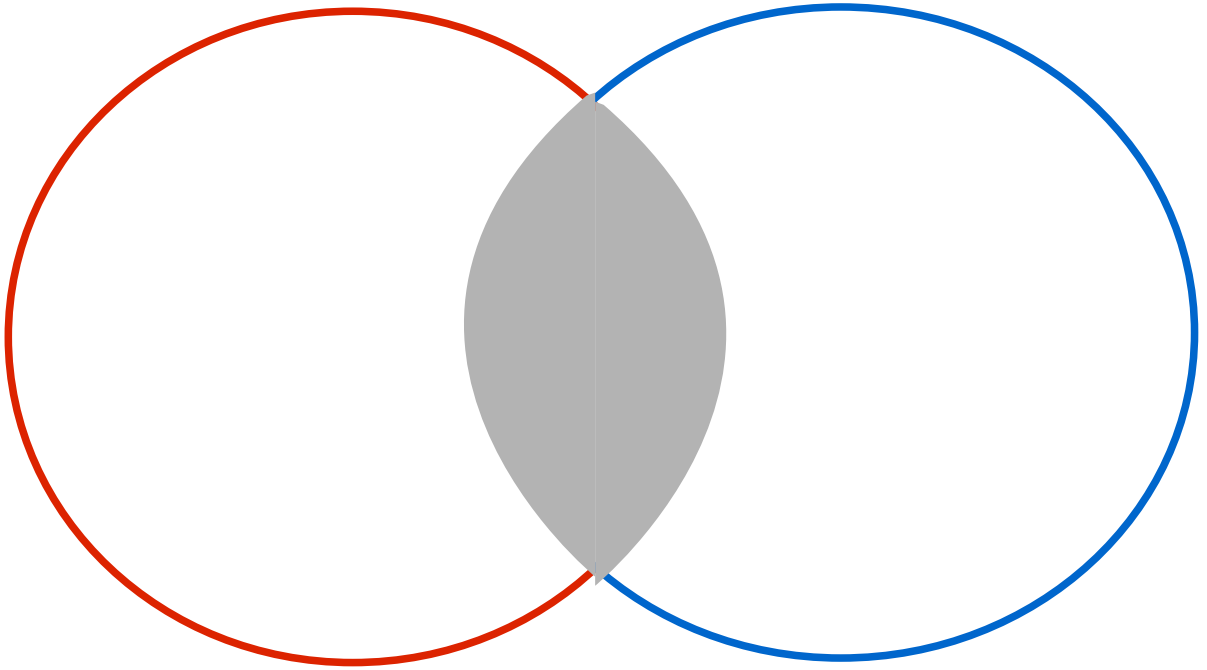
The big idea is "messaging" ...

It's all about messages

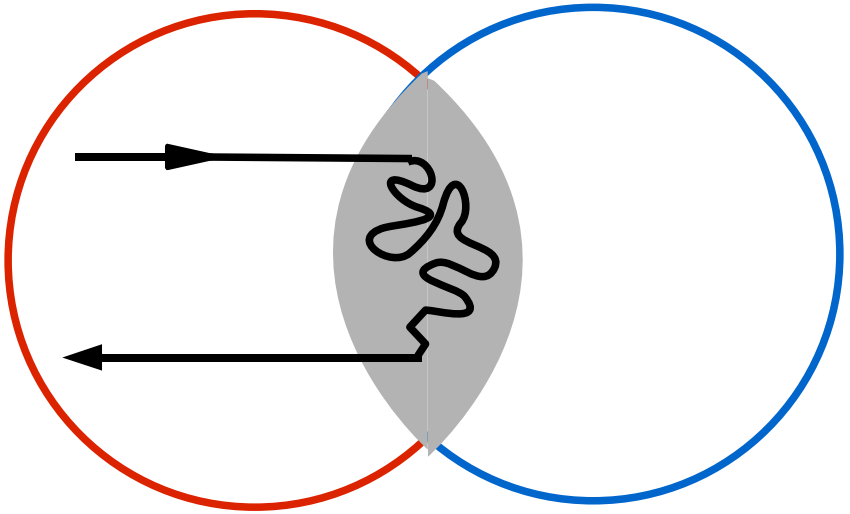
$A \mid B$

Fault-tolerance

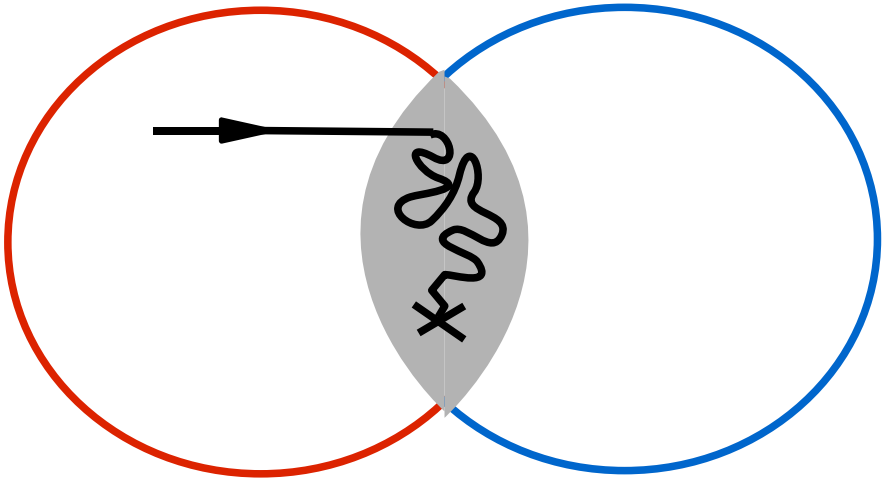
Shared memory



Ooooooooooch

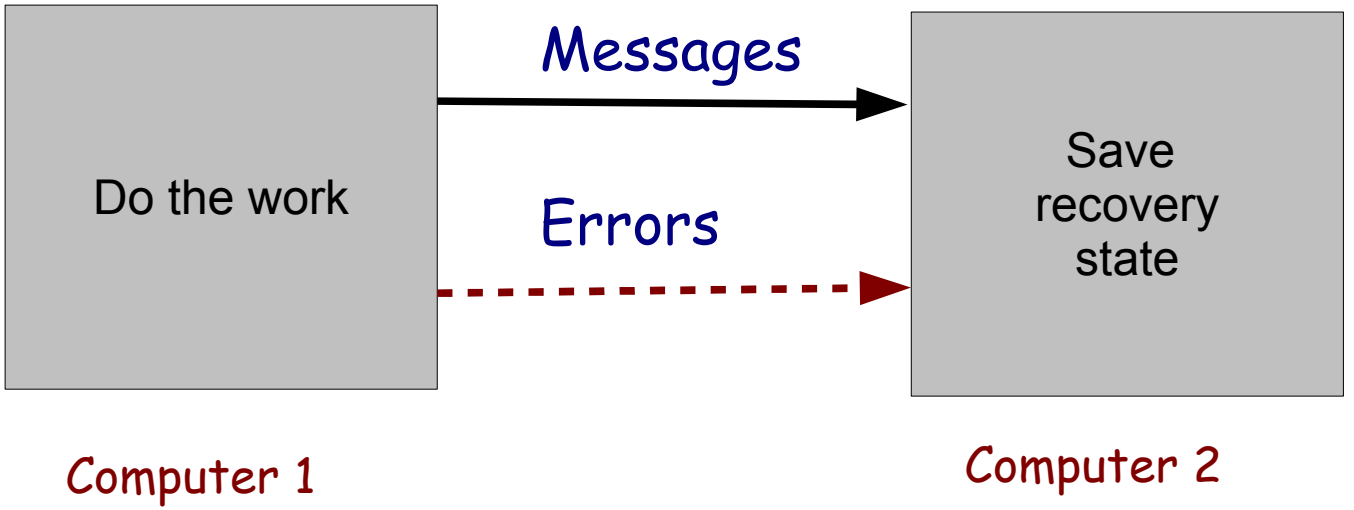


Your program crashes in the critical region having corrupted memory

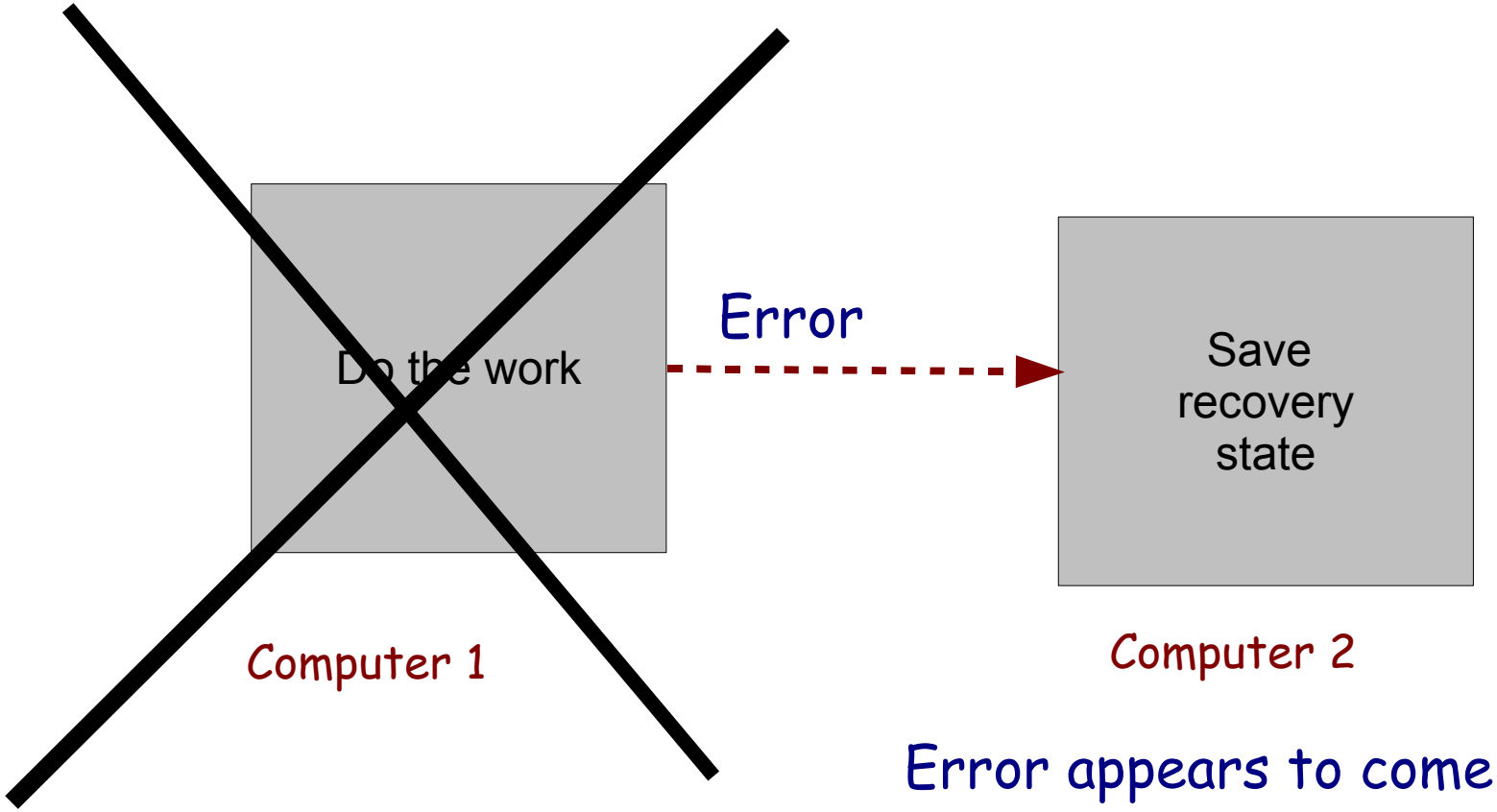


Shared memory and fault tolerance is incredibly difficult
So forbid shared memory

Basic fault-tolerance



Remote error recovery



If machine 1 crashes machine 2 must take over

Error appears to come from machine 1 – in fact a ping monitor on machine 2 detected that machine 1 had failed.

Message-passing Concurrency

How do we think about parallel programs?

Think about?

- Messages (what's in a message?)
- Who knows what?
- Protocols – what are the order of the messages
- What are the processes?

Design

- Identify the processes
- Identify the message channels
- Name the channels
- Name the messages – what are the messages, what's in the messages?
- Specify the message content
- Specify the message order

Q: What can you do with messages?

A: Everything?

Fun with Erlang

- Send
- Receive
- Catch
- Function Application

Send and receive

- Send a message to the mailbox of a process

```
Pid ! Message
```

- Waits for a message that matches a pattern in the mailbox

```
receive
    Pattern ->
        Action
end
```

Servers

Questions

- How do we find the server?
- How do we encode the messages?
- What happens if things go wrong?
- How do we specify the order of messages?

Finding the server

Ipv4 - TCP/IP

213.45.67.23 ! hello

Or

"www.some.host" ! hello

+

DNS

Erlang

Pid ! Hello

Or

some_name ! Hello

And

The process registry

Encoding the message

About 4894 Defined
TCP protocols [1]

```
GET /intro.html HTTP/1.1
Accept: text/html, application:xhtml+xml
...
HTTP 1.1 200 OK
...
```

Erlang
One - Protocol

Pid ! hello

sends

<<131,100,0,5,104,101,108,108,111>>

[1] IANA (Internet Assigned Numbers Authority)
"Well known" Ports

What happens if things go wrong

Ipv4 - TCP/IP

Socket Closed
Or
Hangs

Erlang

```
receive
   {'EXIT', Pid, Why} ->
      ... fix it ...
end
```

An Erlang Server

```
loop(... ) ->
    receive
        {From, Request} ->
                Response = F(Request),
                From ! {self(), Response},
                loop(...)
    end.
```

I'll rewrite this in lot's of different ways

PING

Client

```
Pid ! {self(), ping},
receive
    {Pid, pong} ->
            ... joy ...
 end
```

Server

```
loop() ->
    receive
        {From, ping} ->
            From ! {self(), pong},
            loop()
end.
```

Counter

Client

```
Pid ! {self(), bump},
receive
    {Pid, N} ->
          ...
 end,
```

Server

```
counter(N) ->
    receive
        {From, bump} ->
            From ! {self(), N+1},
            counter(N+1)
end.
```

Generalise the counter

Old

```
counter(N) ->
    receive
        {From, bump} ->
            From ! {self(), N+1},
            counter(N+1)
end.
```

New

```
counter() ->
    loop(0, fun counter/2).

loop(State, F) ->
    receive
        {From, X} ->
            {Reply, State1} = F(X, State),
            From ! {self(), Reply},
            loop(State1, F)
    end.

counter(bump, N) ->
  {N+1, N+1}.
```

Why generalize?

Because we can have some fun

- Send code to the server
- Send data to the server
- Add code upgrade
- Add transactions

Send the code to the server

Client

```
Pid ! {self(),
        fun counter/2},
receive
   {Pid, N} ->
          ...
 End.

counter(N) ->
   {N+1, N+1}.
```

Server

```
loop(State) ->
   receive
      {From, F} ->
          {Reply, State1} = F(State),
          From ! {self(), Reply},
          loop(State1)
end.
```

The sever maintains state – we send code to the server in a message. There is no code on the server

Send the state to the server

```
Pid ! {self(), 10},
receive
    {Pid, N} ->
          ... ...
 end.
```

The server has no data. It stores a function that is applied to data that comes from the client

```
counter() ->
    loop(fun counter/1).

loop(F) ->
    receive
        {From, State} ->
            Reply = F(State),
            From ! {self(), Reply},
            loop(F)
end.

counter(N) ->
    N+1.
```

Code Upgrade

```
rpc(Pid, N) ->
    Pid ! {self(), Q},
    receive
        {Pid, R} -> R
    End.

triple(X) -> X*X*X.

> rpc(Pid, 2).
4
> Pid ! {upgrade,fun triple/1}.
...
> rpc(Pid, 2)
8
```

```
start() ->
     loop(fun  double/1).

loop(F) ->
    receive
        {upgrade, F1} ->
            loop(F1);
        {From, X} ->
            Reply = F(X),
            From ! {self(), Reply},
            loop(F)
    end.

double(X) -> 2*X.
```

Code Upgrade with state

```
start() ->
    loop(State, fun doit/2).

loop(State, F) ->
   receive
      {upgrade, F1} ->
         loop(State, F1);
      {From, X} ->
          {Reply, State1} = F(X, State),
          From ! {self(), Reply},
          loop(State1, F)
   end.

doit(X, State) -> .... {Reply, State1}.
```

Code Upgrade with state upgrade

```
start() ->
    loop(State, fun doit/2).

loop(State, F) ->
   receive
      {upgrade, F1, F2} ->
         State1 = F2(State),
         loop(State1, F1);
      {From, X} ->
          {Reply, State1} = F(X, State),
          From ! {self(), Reply},
          loop(State1, F)
   end.

doit(X, State) -> .... {Reply, State1}.
```

Were you watching carefully?

- loop() - (PING)
- loop(State, Fun) – stateful server
- loop(State) – mobile code
- loop(Fun) – mobile data

- Can we generalize the generalizations?

The Universal Server

```
wait() ->
    receive
        {become, F} ->
                F()
    end.
```

So let's let the client send the server code to the server

```
Pid ! {become, fun() -> loop(fun(Id) -> Id end) end}.

loop(F) ->
  receive
     {upgrade, F1} ->
        loop(F1);
     {From, X} ->
        Reply = F(X),
         From ! {self(), Reply},
         loop(F)
  end.
```

What have we done?

Traditional

- TCP/IP
- 4894 ad hock protocols
- Implement a server for ONE of them (repeat 4894 Times)
- Allow plugins (example Apache)

Erlang

- One protocol
- One Generic Server
- The application (say an HTTP server **is** the plugin)

Observations

- Conventionally servers mainatain state
- Conventionally we move the data to the computation (example, mysql, the data-base has the data, the data is moved to the client where the computation is performed)
- We can move the data, or the computation, whichever is most effective
- No locks – or classes – just messages

Behaviours

- Collect the powerful generalisations
- Give them names
- Document their usage

  (write a few millions of line of code that use them to see if they work – they do)

6 Behaviors

- gen_server
- gen_fsm
- supervisor
- gen_event
- aspplication
- release

Where does the power come from?

- Dynamic (safe) types
  binary_to_term/term_to_binary
- One encoding (slide + 2)
- Late Binding
- Higher order

  Functions are data – can send functions in messages
- Pure MP
- Easy to invent abstractions
- No destructive assignment (slide + 3)

One Encoding

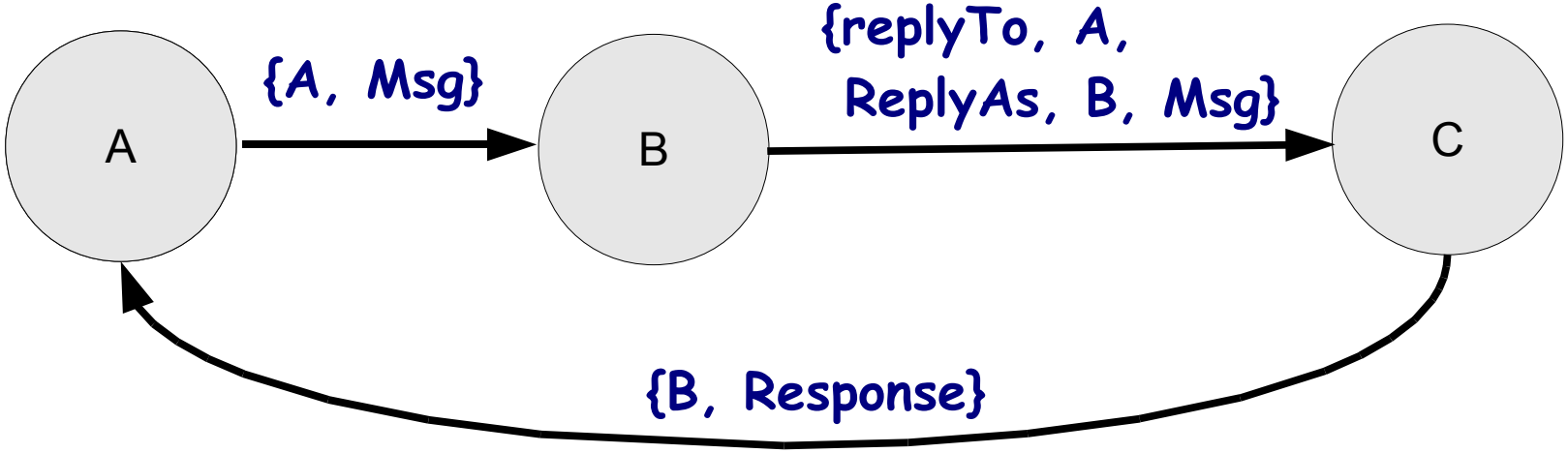
Email + FTP (HTTP) + IM (the power of one protocol)

```
loop() ->
    receive
        {email, _From, _Subject, _Text} = Email} ->
            {ok, S} = file:open("inbox", [write,append]),
            io:write(S, Email),
            file:close(S);
        {im, From, Text} ->
            io:format("Msg (~s): ~s~n",[From, Text]);
        {Pid, {get, File}} ->
            Pid ! {self(), file:read_file(File)}
    end,
    loop().
```

Now let's add transactions

```
loop(State, F) ->
  receive
     {From, X} ->
        case (catch F(X, State)) of
           {'EXIT', Why} ->
               From ! {self(), {error, Why}},
               loop(State, F);
           {Reply, State1} ->
               From ! {self(), {ok, Reply}},
               loop(State1, F)
  end.
```

Client server is only one pattern there are many more



A ! B

in

more detail

IMPORTANT

A ! B

enforces isolation

must be asynchronous

A ! B glues things together

TEXT TEXT TEXT

```
$ find .. | grep "module" | uniqu | wc
```

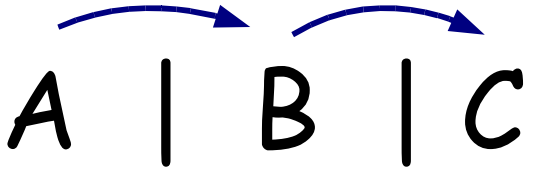
+ each component can be in a different language
- Text flows across the boundaries = lots of parsing/formatting

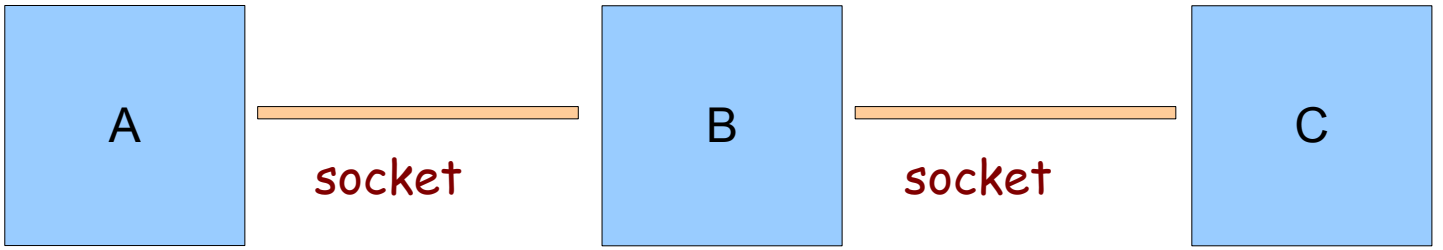
The output of your program might one day be the input to somebody elses Program

Structured term

```
Pid ! Msg  | receive Pattern -> Action end
```

A ! B makes distribution posible





A ! B is great

but

what is A?

A is ...

- A process
- A mailbox

The mailbox

RFC 196 (July 1971)
A mail box, as we see it,
is simply a sequential file to
Which messages and documents
are appended, separated by
an appropriate
site dependent code.

RFC 821 (Postel) (August 1982)

S: MAIL FROM:<Smith@Alpha.ARPA>
R: 250 OK

Mailboxes

- Send and receive is decoupled
- Location transparent (send to a name not a location)
- Messages stay in mailbox until read
- Reliable/Secure/Order preserving?

Message passing architectures are everywhere

tcpmux compressnet rje echo discard systat daytime qotd msp chargen ftp-data ftp ssh telnet smtp nsw-fe msg-icp msg-auth dsp time rap rlp graphics name nicname mpm-flags mpm mpm-snd ni-ftp auditd tacacs re-mail-ck la-maint xns-time dns xns-ch isi-gl xns-auth xns-mail ni-mail acas whois++ covia tacacs-ds sql*net bootps bootpc tftp gopher netrjs-1 netrjs-2 netrjs-3 netrjs-4 deos vettcp finger http hosts2-ns xfer mit-ml-dev ctf mfcobol kerberos su-mit-tg dnsix mit-dov npp dcp objcall supdup dixie swift-rvf tacnews metagram newacct hostname iso-tsap gppitnp acr-nema csnet-ns 3com-tsmux rtelnet snagas pop2 pop3 sunrpc mcidas ident audionews sftp ansanotify uucp-path sqlserv nntp cfdptkt erpc smakynet ntp ansatrader locus-map nxedit locus-con gss-xlicen pwdgen cisco-fna cisco-tna cisco-sys statsrv ingres-net epmap profile netbios-ns netbios-dgm netbios-ssn emfis-data emfis-cntl bl-idm imap uma uaac iso-tp0 iso-ip jargon aed-512 sql-net hems bftp sgmp netsc-prod netsc-dev sqlsrv knet-cmp pcmail-srv nss-routing sgmp-traps snmp snmptrap cmip-man cmip-agent xns-courier s-net namp rsvd send print-srv multiplex cl/1 xyplex-mux mailq vmnet genrad-mux xdmcp nextstep bgp ris u udit ocbi er remot mumps qft gacr pero osu-nms srmp irc dn6-nlm n6-smm dis d n sm at at-nbp at-3 a at-5 at-zis at-7 at-8 qmtp z39.5 anet i mpwsc tpc c dbase uarps im x rsh-spx cdc masqdialer d neas in ness dsp3 subntb tp bhfh chat esro-gen openport n s a ms hdap one-ctl td-serv -replic p-r personal-link cableport- escap erjd fxp ast orb ne b s asi badmin vslmp magenta pkix-tim eve ip rtsps texar pdap pa c wp mftp p-type-a bhoett ed datex-asn cloanto-net-1 bnev hrinkw srmp s odial man send rsvp_tunn urora-cmgr dtk odmr mortgagewar ikgdp rp odaauth ulistproc legent gent-2 hassle nip tnETOS dsETOS is99c is99s hp-collector hp-managed-node hp-alarm-mgr arns ibm-app asa aurp unidata-ldm ldap uis synotics-relay synotics-broker meta5 embl-ndt netcp netware-ip mptn kryptolan iso-tsap-c2 work-sol ups genie decap nced ncld imsp timbuktu prm-sm prm-nm decladebug rmt synoptics-trap smsp infoseek bnet silverplatter onmux hyper-g ariel1 smpte ariel2 ariel3 opc-job-start opc-job-track icad-el smartsdp svrloc ocs_cmu ocs_amu utmpsd utmpcd iasd nnsp mobileip-agent mobilip-mn dna-cml comscm dsfgw dasp sgcp decvms-sysmgt cvc_hostd https snpp microsoft-ds ddm-rdb ddm-dfm ddm-ssl as-servermap tserver sfs-smp-net sfs-config creativeserver contentserver creativepartnr macon-tcp scohelp appleqtc ampr-rcmd skronk datasurfsrv datasurfsrvsec alpes kpasswd urd digital-vrc mylex-mapd photuris rcp scx-proxy mondex ljk-login hybrid-pop tn-tl-w1 tcpnethaspsrv tn-tl-fd1 ss7ns spsc iafserver iafdbase ph bgs-nsi ulpnet integra-sme powerburst avian saft gss-http nest-protocol

MPC is great

- Shared state + Errors is ~~impossibly~~ difficult to understand
- Pure messaging is built into the fabric of the universe – messages are bundles of photons
- Message passing is the most common way to program distributed systems

  but we need a substrate ...

Message passing substrates

- Persistent named job queues
- If you want a job done you send a message to a named queue. It will eventually be done, and you will get a message back
- Decouples processing from the job queue
- All data needed to do the job is in the message itself

Some Message passing substrates

- Erlang

  Very fast – in memory – volatile

- MPI

  Very fast - Industry standard message passing library

- KILIM

  Message passing library for JAVA

- Email

  Slow – non-volatile – no guarantees

- AMQP

  Medium – reliable storage - guarantees

Benefits

- Same model works for programming in-the-small and in-the-large
- Functional – Output depends only upon the inputs
- Scalable
- Reliable

The Erlang Experience

- Efficient
- Scales for very large systems. Copying overhead "not a problem"
- High reliability is possible
- Multi-core ready (here-and-now)
- Works in large S/W projects (> 1 million lines of code)
- Used in many "core" Internet applications
- Plays well with other languages (but not in memory)

Not the End

What's Missing?

It's all about protocols

- 4894 TCP protocols – each one has it's own syntax
- Need a type system and "protocol types" - something like CSP, Pi calculus, UBF, ...
- We have no good way of describing protocols

The End