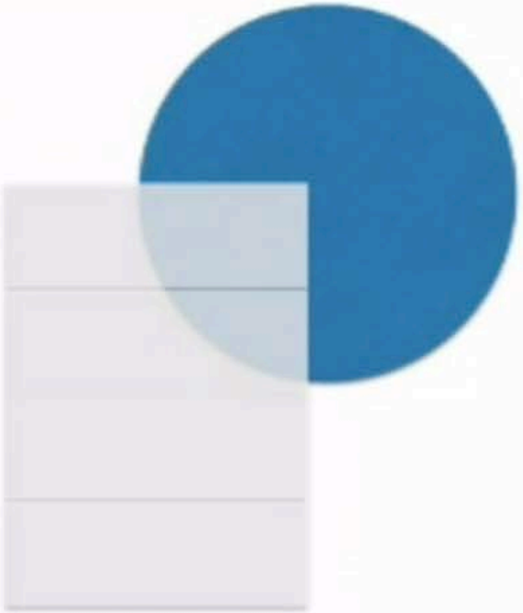
University of Kent

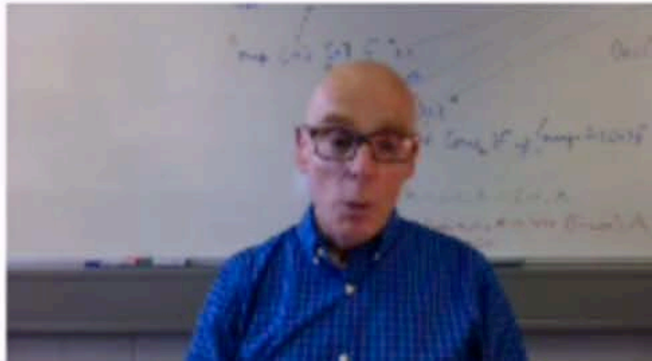
The mailbox in Erlang

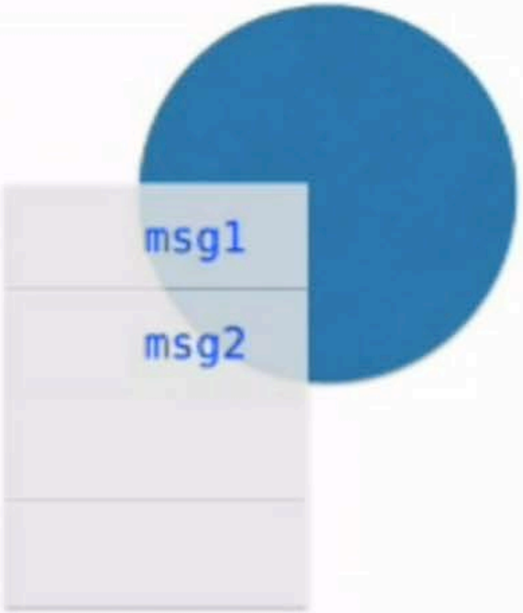
Erlang concurrency in a nutshell

`spawn`	– create a process
`!`	– send a message
`receive`	– process a message
`self()`	– give the Pid of a process



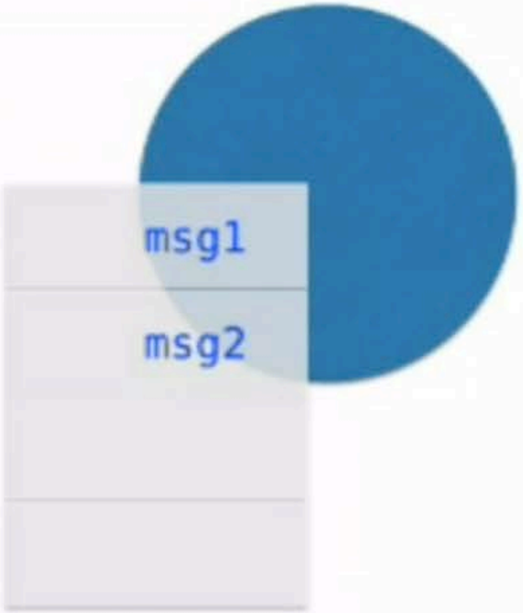
```
Pid ! msg1,
Pid ! msg2,
  …
```





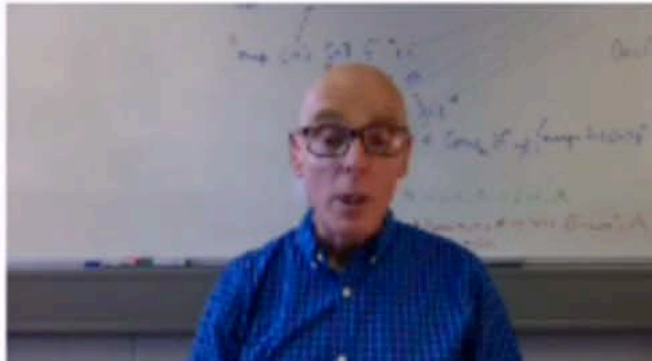

```
…
```

```
receive
  Msg -> …
end
```




```
…
```

```
receive
  msg2 -> …
end
```



University of Kent

How does message passing work?

When a message is sent to a process, it is added to the mailbox of the process …

… and messages are stored in the order that they arrive.

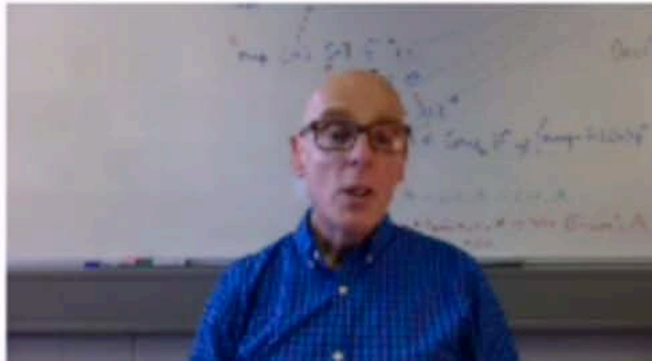
Messages in the mailbox are then handled by `receive` statements.

The details of `receive`

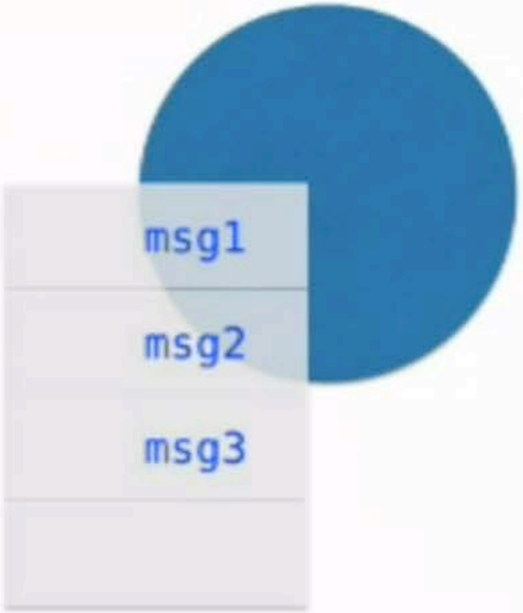
Messages *do not* have to be handled in mailbox order.

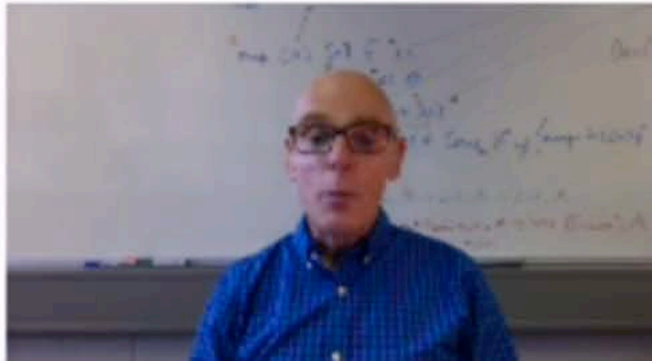
Try the clauses of the `receive` in turn on the first message …
… and apply the first matching clause;
… if no clause matched, go to the second message and repeat …
… if no message matches any clause, wait for an incoming message.

Scenario 1



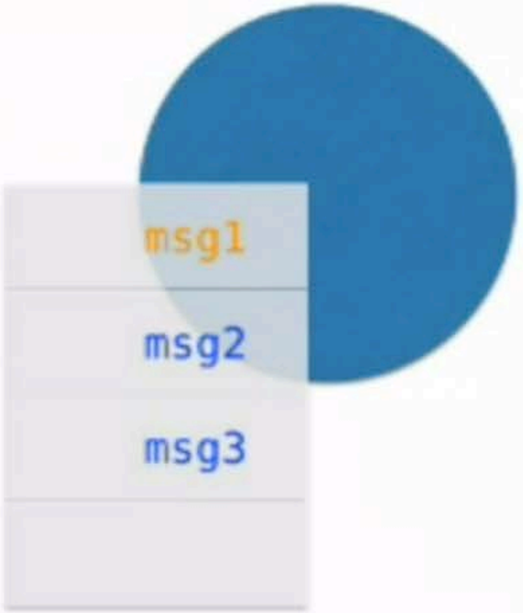
```
receive
  msg2 -> …;
  Msg  -> …;
  …
end
```



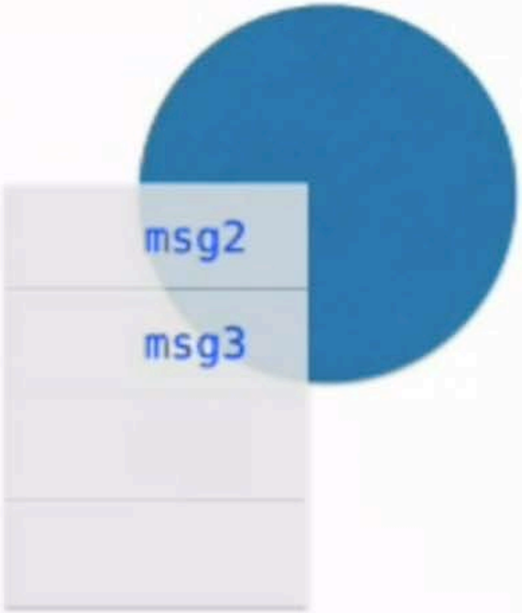


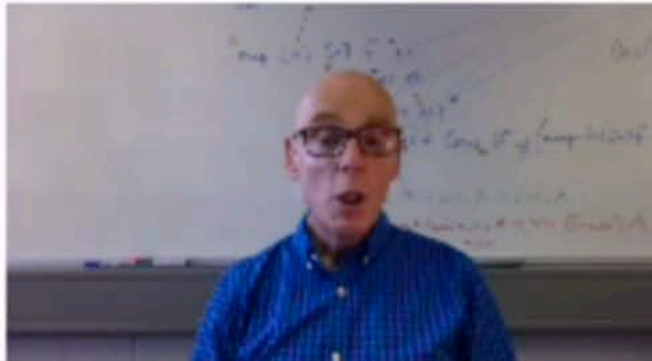
```
receive
  msg2 -> …;
  Msg  -> …;
  …
end
```

```
receive
  msg2 -> …;
  Msg  -> …;
  …
end
```



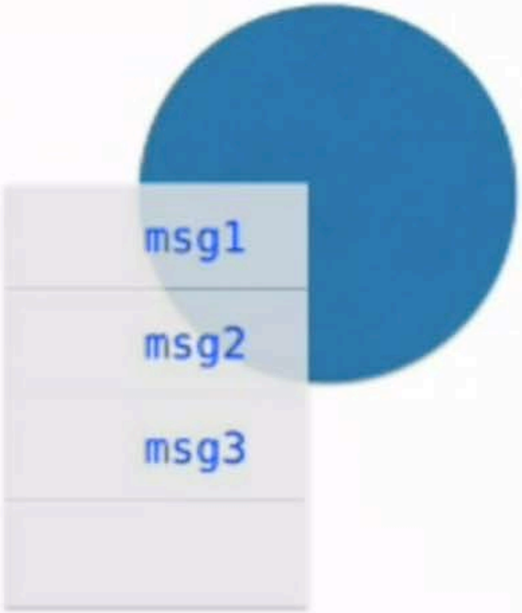
...


msg2
msg3

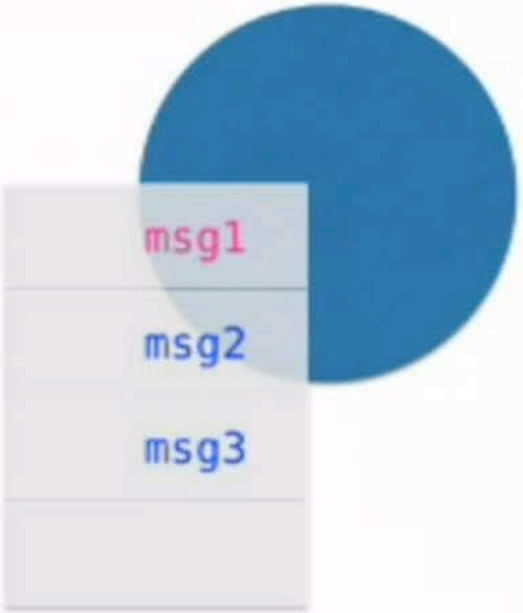


Scenario 2

```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```

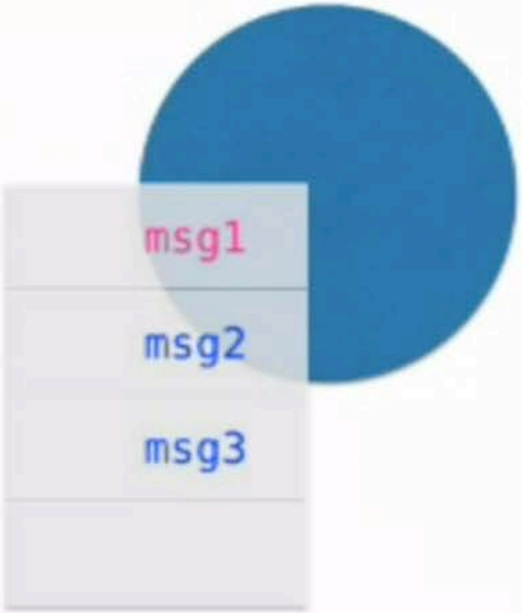


```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```

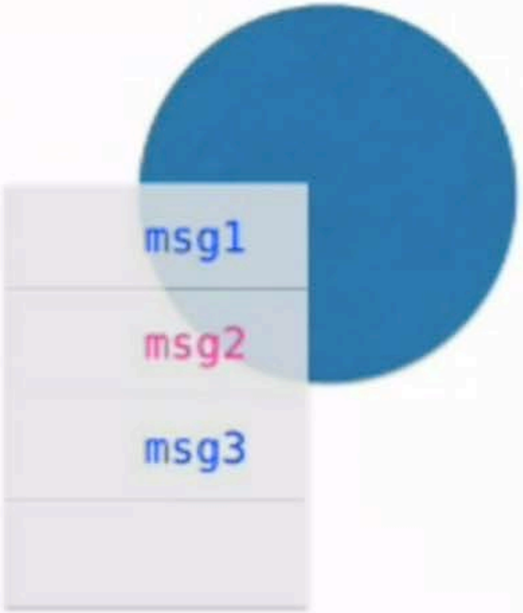


```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```

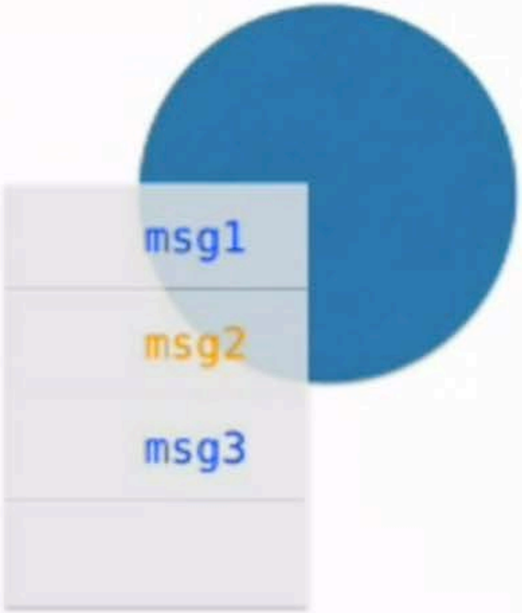
```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```

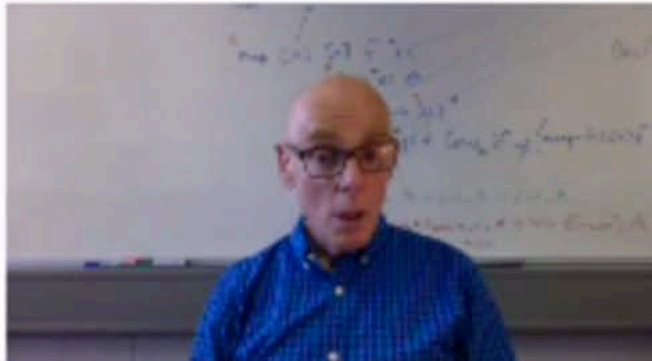


```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```

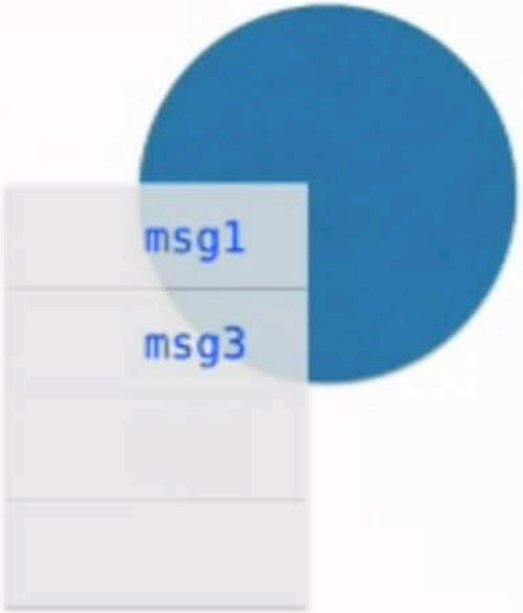


```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```





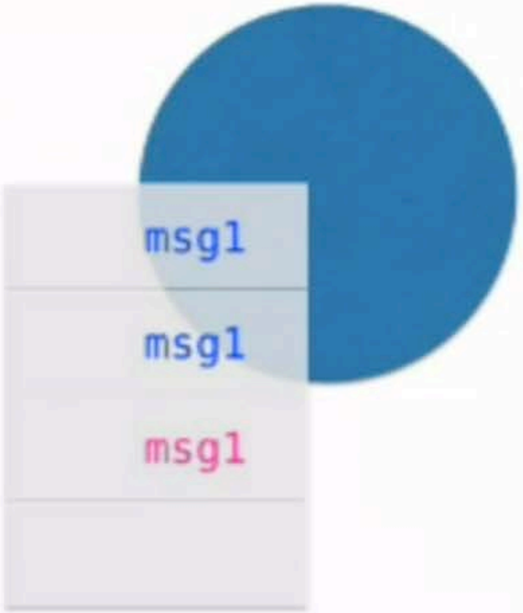
...


msg1
msg3

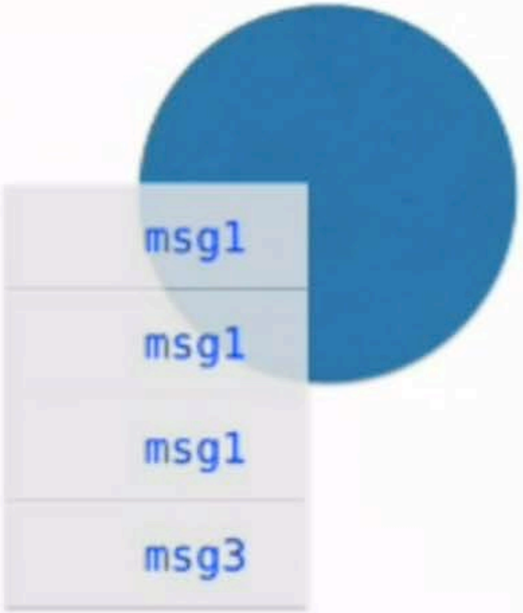
Scenario 3

```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```

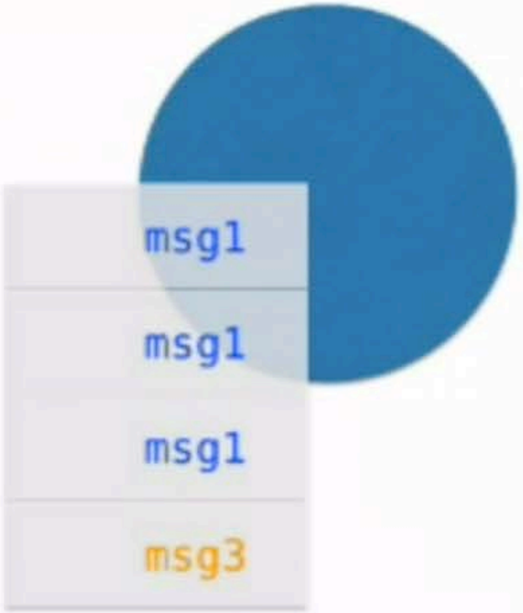
```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```



```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```



```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```



...


msg1
msg1
msg1

University of Kent

Handling the mailbox

Message sending is asynchronous …

…and decoupled from message handling using `receive`

Message `receive` is *selective* …

… allowing messages to be handled in the order that we choose.

One last word ...

Just as in the case statement ... in a `receive` it's possible to pattern match against bound variables.

For example, if `Pid` is already bound then

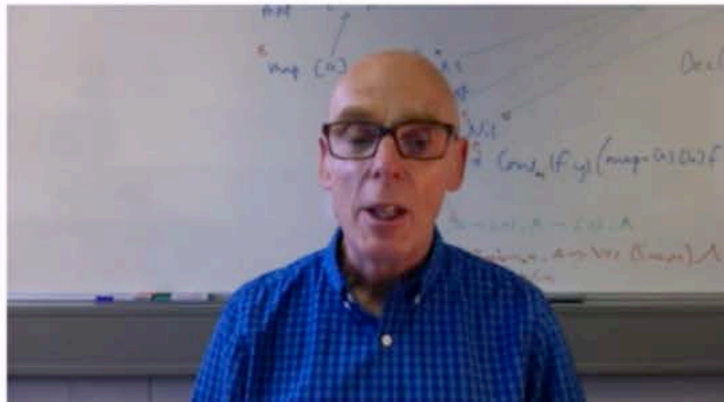
```
receive
 {Pid,Msg} -> …
end
```

will only match messages with that value as first component.

University of Kent

University of Kent

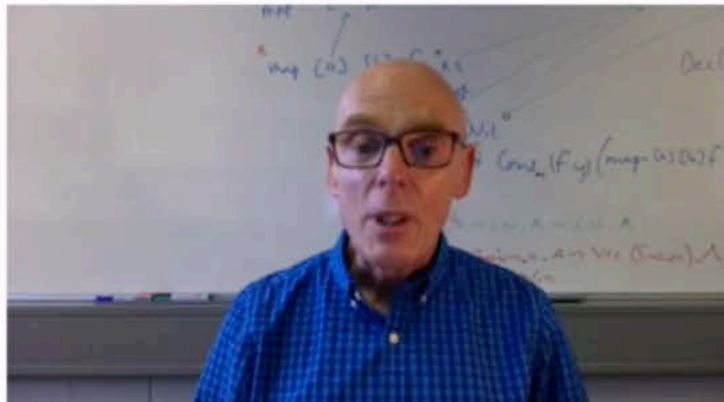
Timing and message ordering

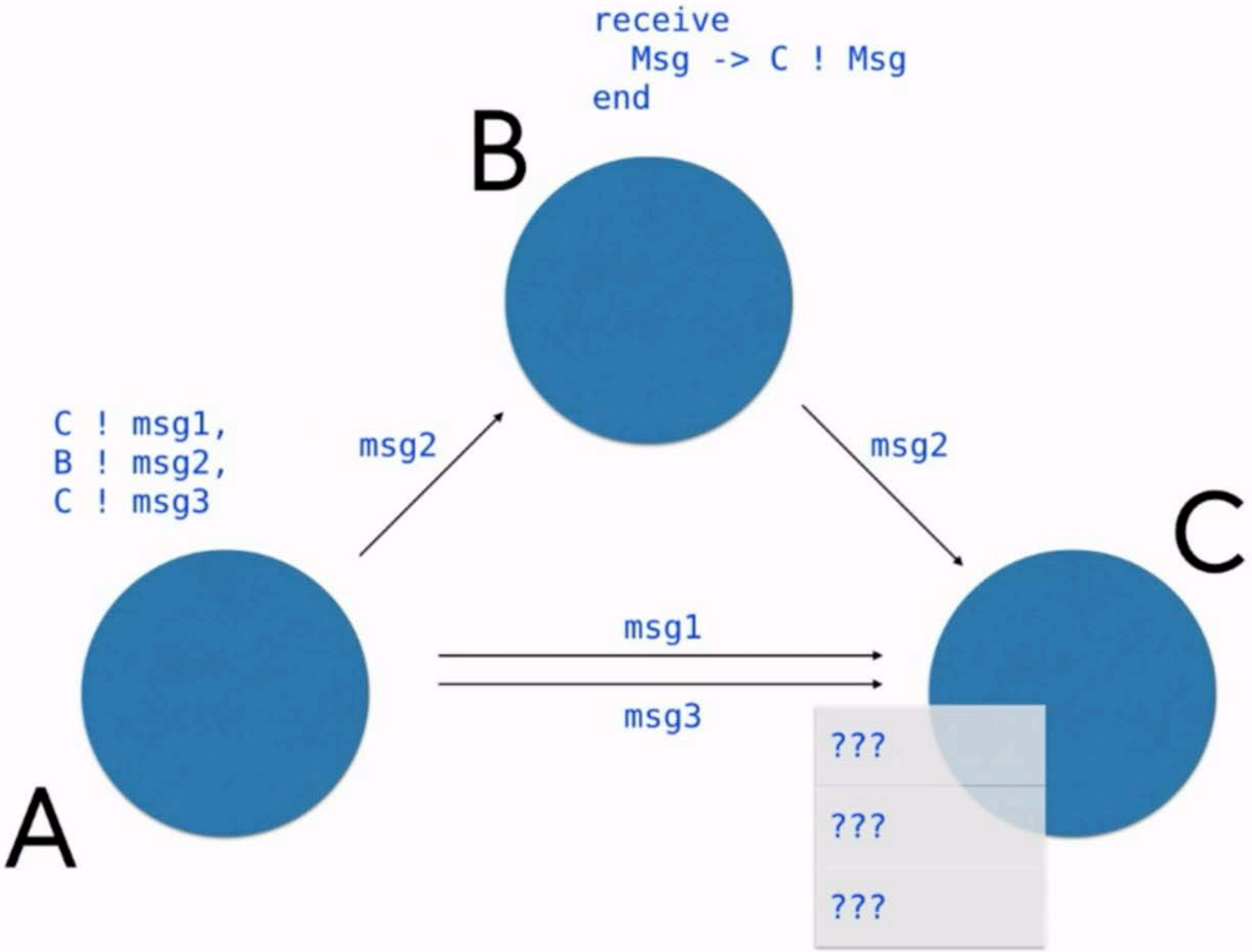


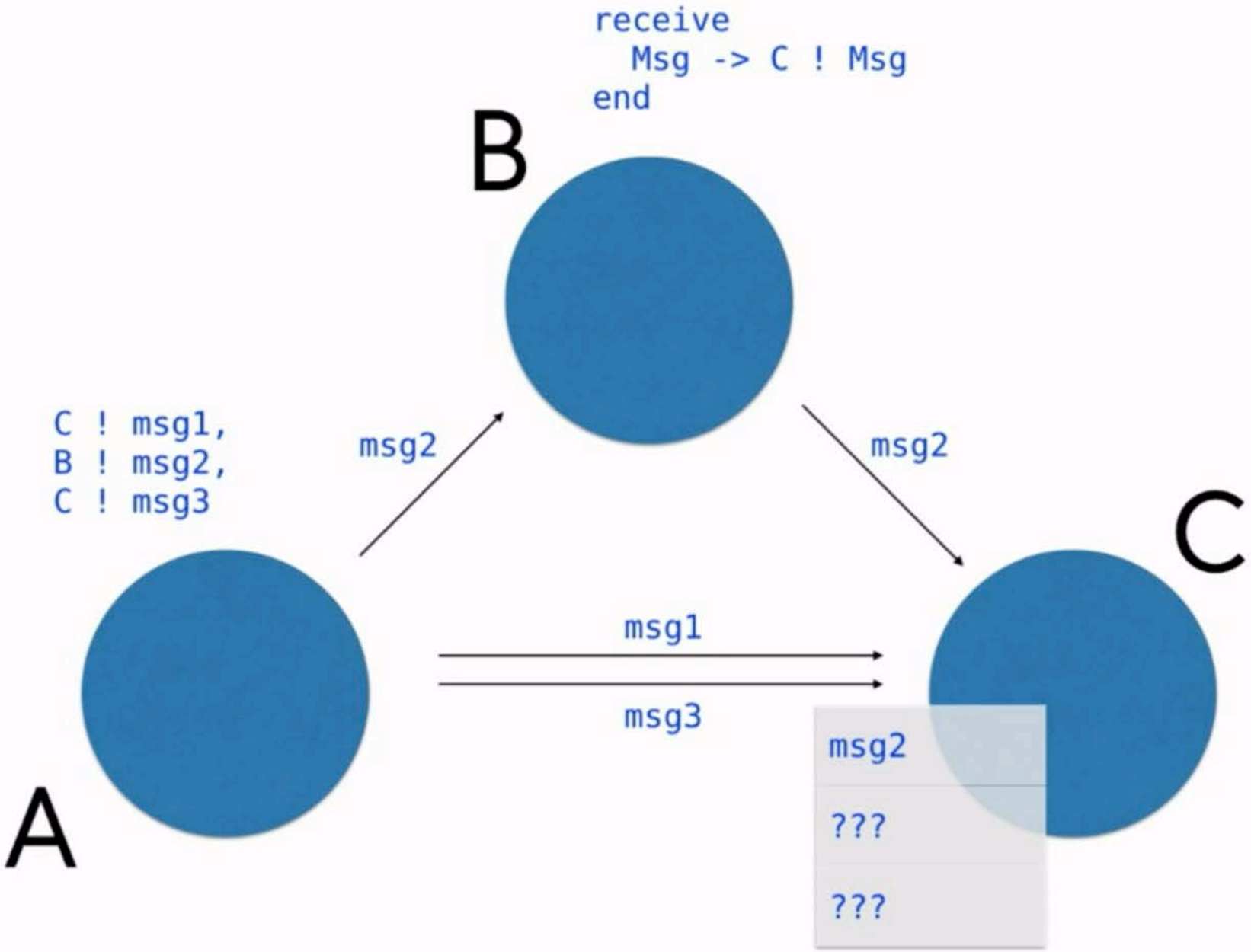
Erlang promise on message ordering

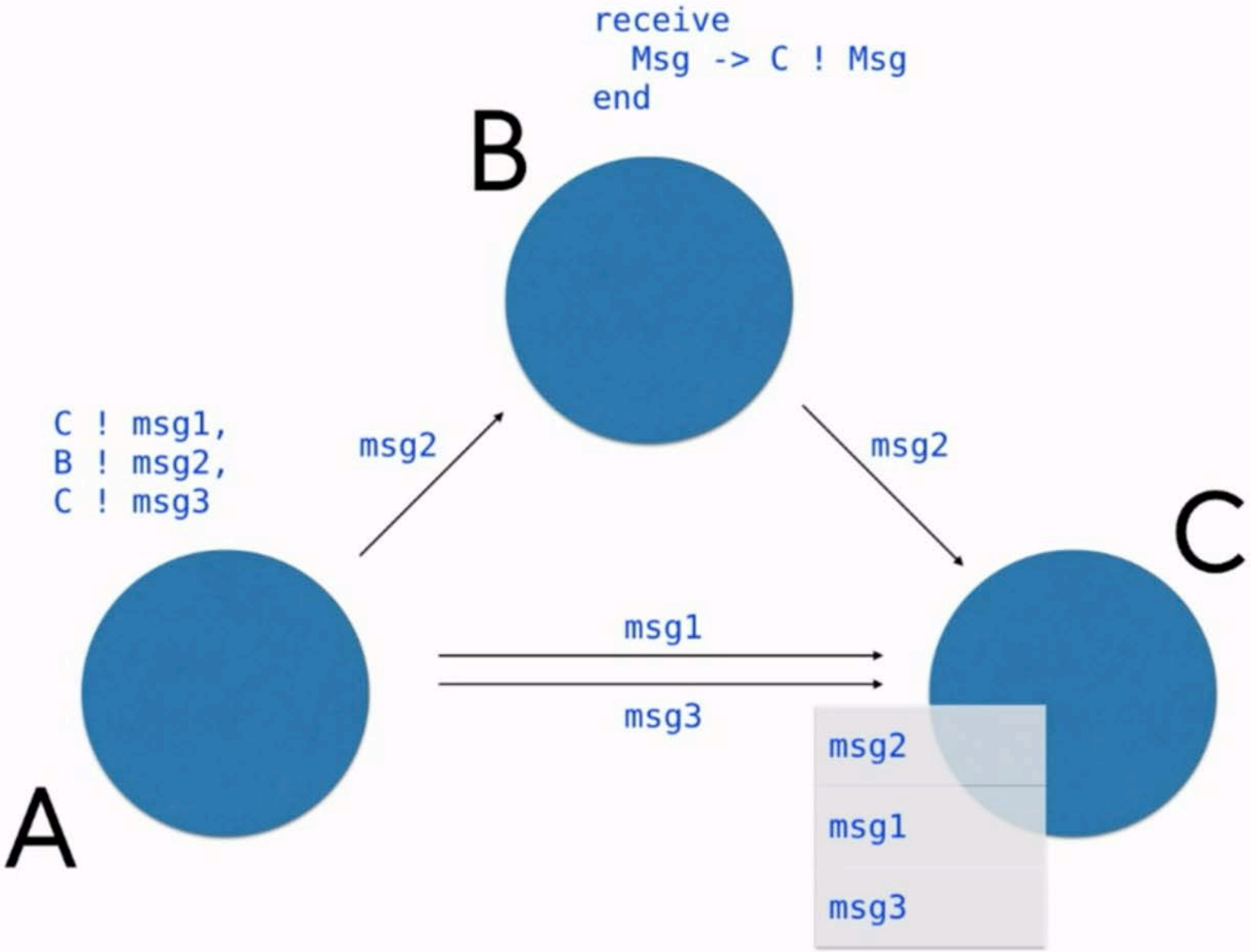
The messages sent between two particular processes will be delivered into the mailbox in the same order that they are sent.

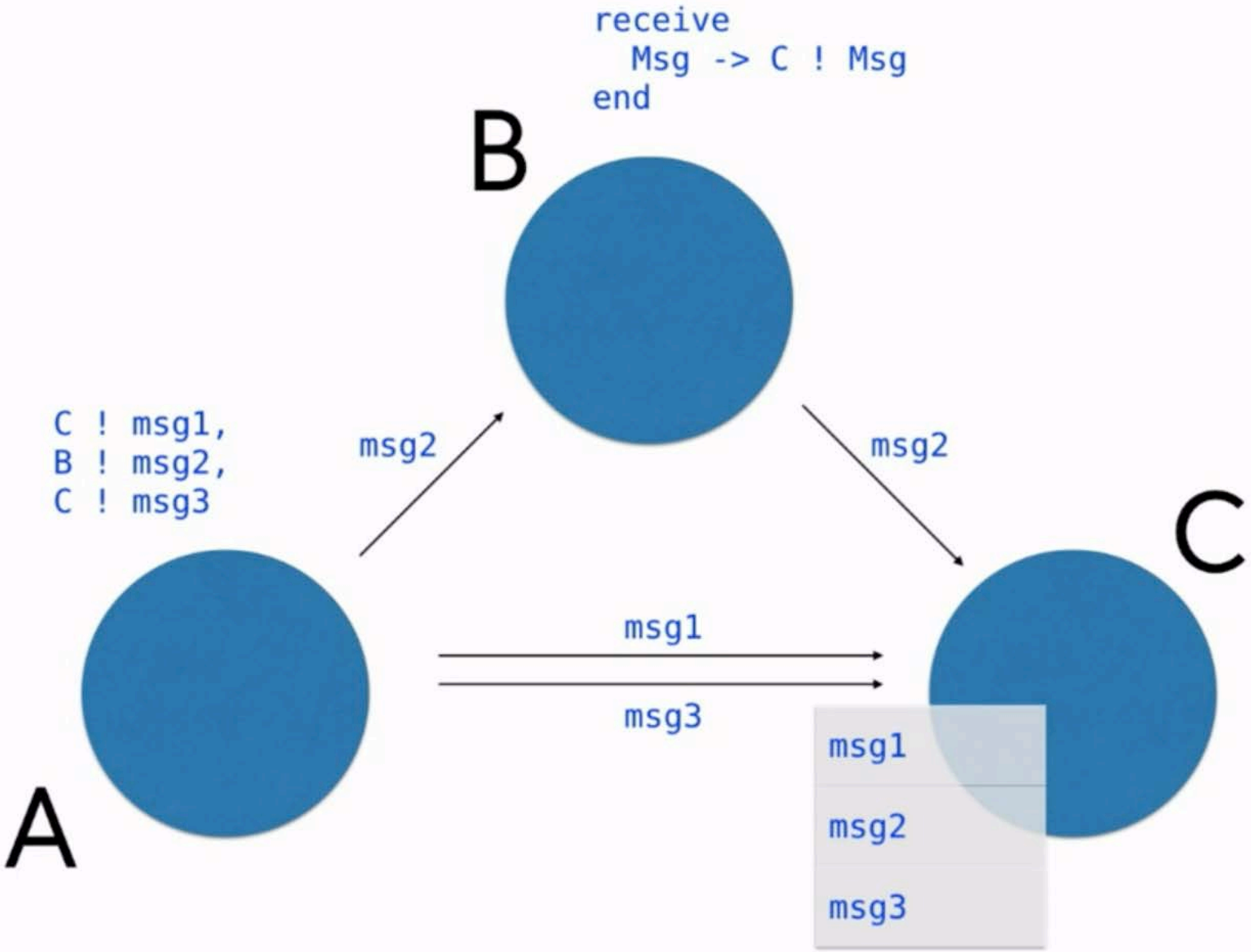
That doesn't cover indirect messaging …

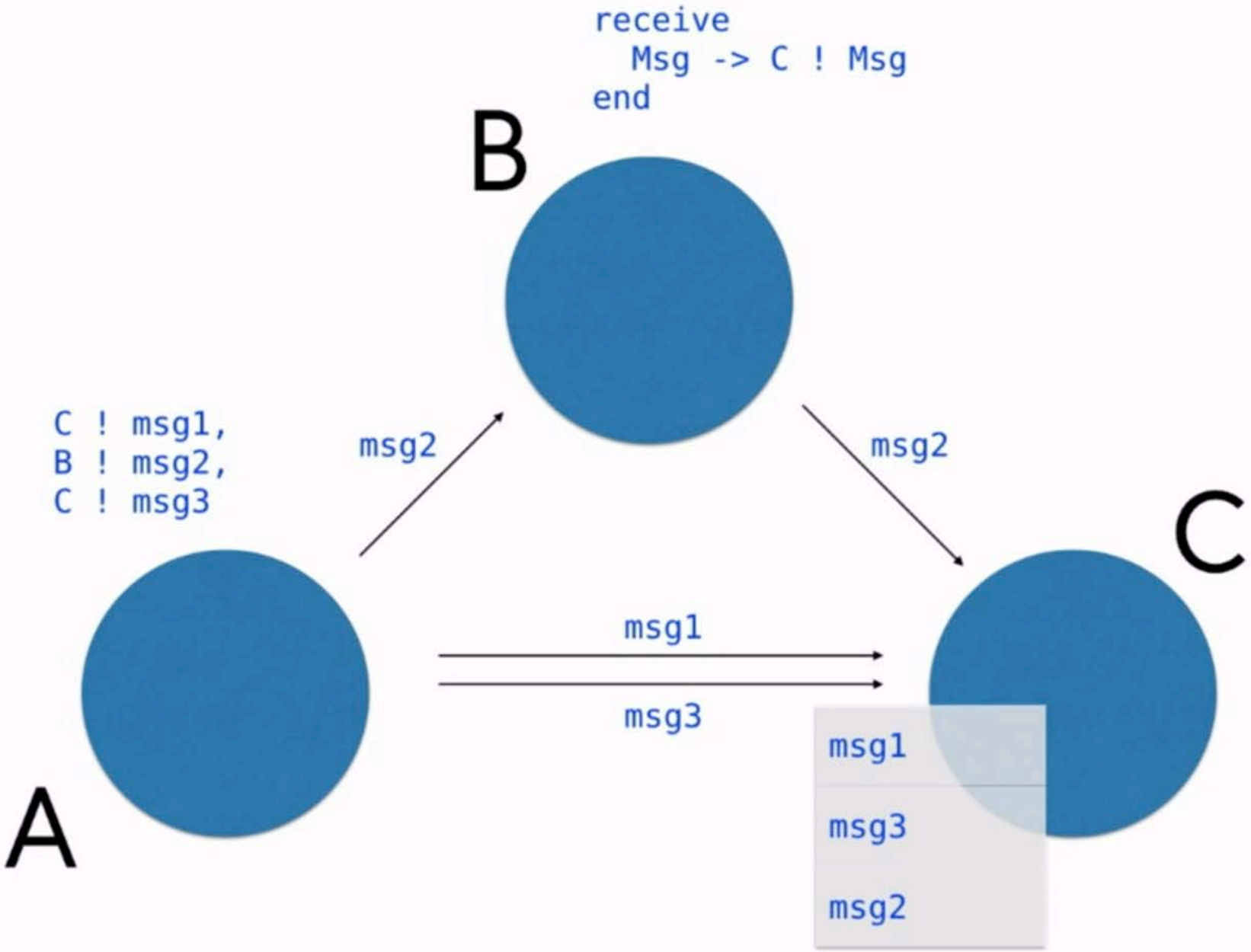



receive
Msg -> C ! Msg
end
B
C ! msg1,
B ! msg2,
C ! msg3
msg2
msg2
C
msg1
msg3
???
???
???
A


receive
Msg -> C ! Msg
end
B
C ! msg1,
B ! msg2,
C ! msg3
msg2
msg2
C
msg1
msg3
msg2
???
???
A


receive
Msg -> C ! Msg
end
B
C ! msg1,
B ! msg2,
C ! msg3
msg2
msg2
C
msg1
msg3
msg2
msg1
msg3
A


receive
Msg -> C ! Msg
end
B
C ! msg1,
B ! msg2,
C ! msg3
msg2
msg2
C
msg1
msg3
msg1
msg2
msg3
A


receive
Msg -> C ! Msg
end
B
C ! msg1,
B ! msg2,
C ! msg3
msg2
msg2
C
msg1
msg3
msg1
msg3
msg2
A

Why should this be?

Recall that on a single processor, at least, processes are scheduled in and out, allowing them to do some work – measured by function reductions.

This has the effect of sequentialising things that might be more concurrent were the resources to be available.

Real behaviour can therefore be less determinate than might be expected.

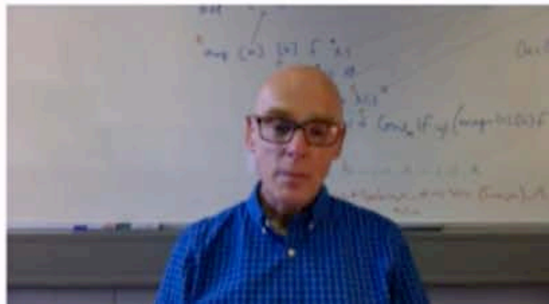
University of Kent

Why is this a good thing?

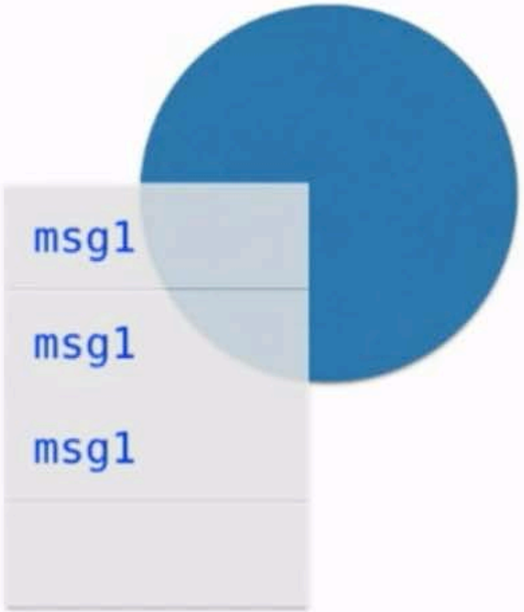
When we come to look at distributed Erlang, we'll see that it has exactly the same behaviour.

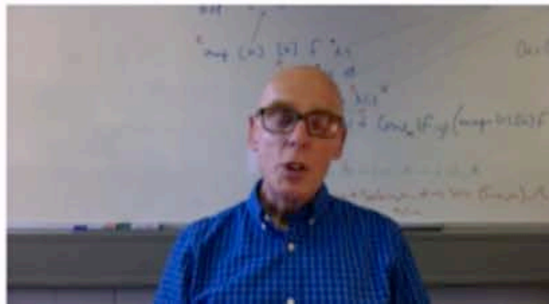
We don't distinguish between local and remote processes.

Timeouts

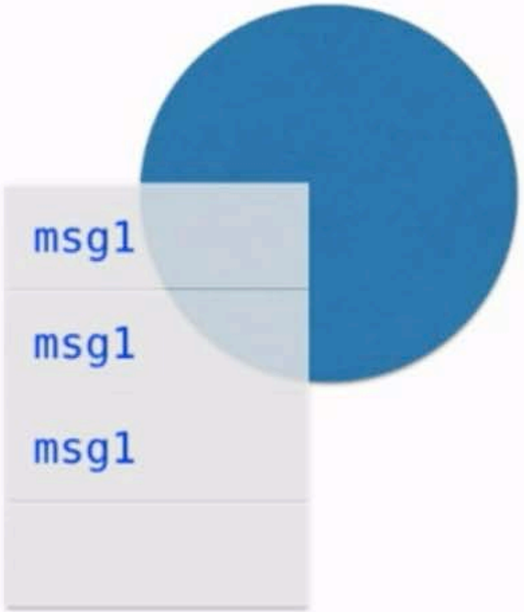


```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
end
```





```
receive
  msg2 -> …;
  msg3 -> …;
  msg4 -> …
after 500 -> …
end
```



Deadlock avoidance – timeout

If none of the patterns can be matched, then after waiting 500 milliseconds perform the action following the `after 500` clause.

What is the potential problem with this?

The message(s) may arrive late, and be in the mailbox later in the program.

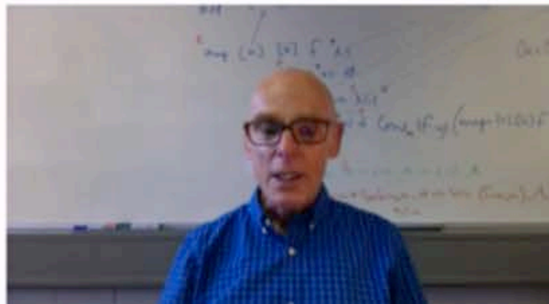
Flushing the mailbox

How can we remove all the messages that are in the mailbox?

```
clear() ->
   receive ->
     Msg -> ok
   end,
   clear().
```

What is the problem with this?

It never terminates … it waits for messages, reads and repeats!



Flushing the mailbox

How can we remove all the messages that are in the mailbox?

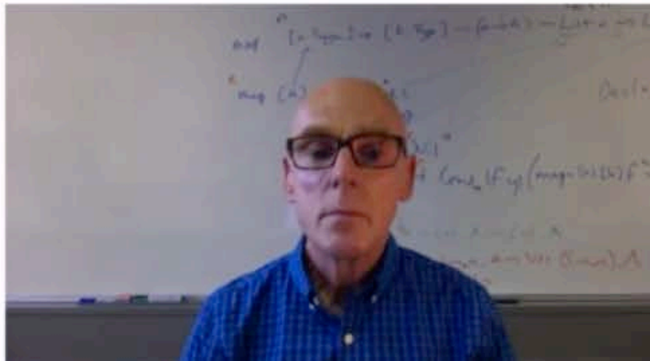
```
clear() ->
  receive
    _Msg -> clear()
  after 0 ->
    ok
  end.
```

If a message is present, read it and repeat … if not (`after 0`) then terminate.

University of Kent

University of Kent

Establishing communication: Pids and named processes

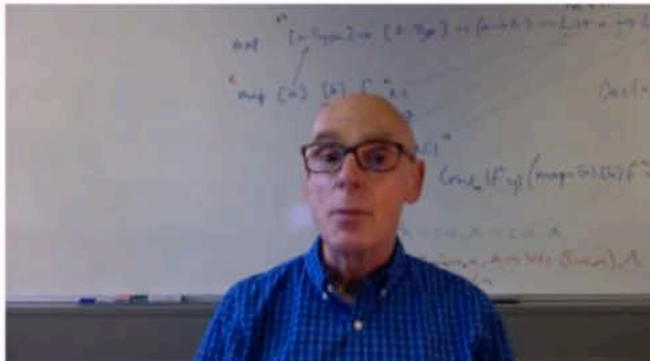


Sending messages

`Pid ! Message` will send the `Message` to the process with id `Pid`.

For us to be able to do this, we need to know the `Pid` …

… and so this needs to be communicated to us somehow.



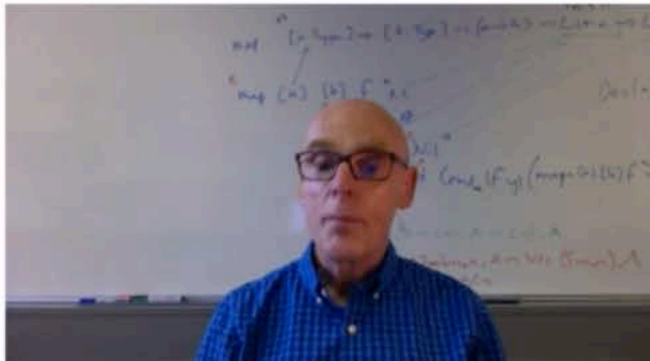
Include your `Pid`

The typical pattern is to send a message to a (`Server`) process with your `Pid` …

```
Server ! {self(), ping},
receive
  pong -> …
end
```

…and your recipient can then send a message back to you.

```
receive
  {Pid, ping} ->
    Pid ! pong
end
```



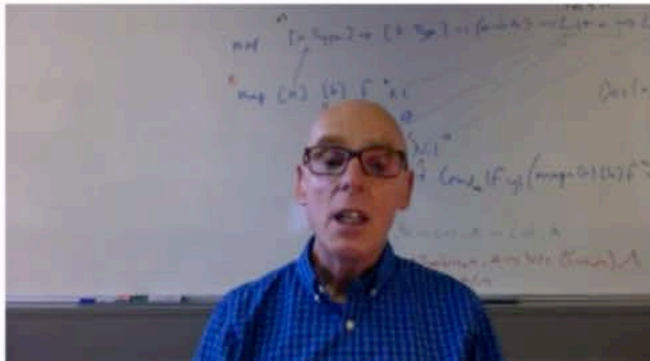
Include your `Pid`

It's also possible to include a Pid in the return message …

… and the client can check that the message has come from the right place.

```
Server ! {self(), ping},
receive
  {Server, pong} -> …
end
```

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



Named processes

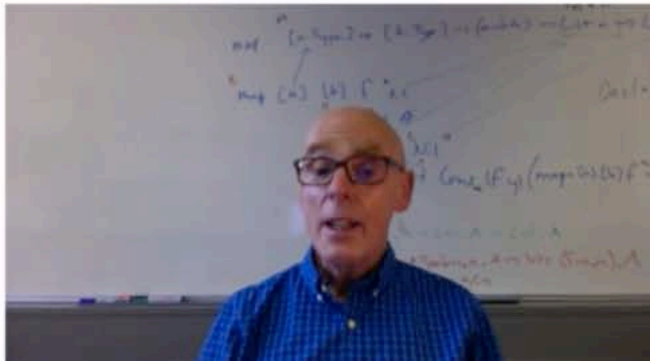
As an alternative we can name a process, e.g. `server` …

```
Server = spawn(M,F,[]),
register(server,Server),
server ! {self(), ping}
```

… and send messages to that name: `server ! Msg`.

Typically `spawn` and `register` are done in one nested call …

```
register(server,spawn(M,F,[])),
```



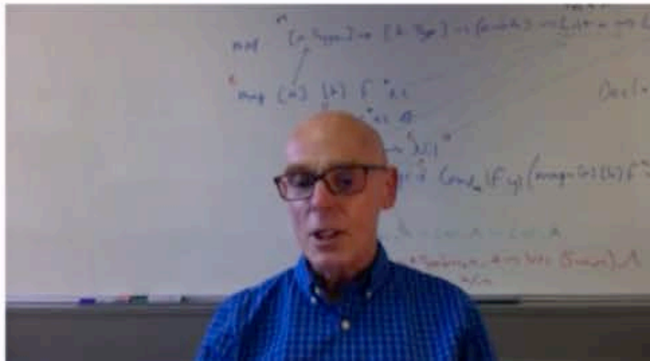
Names *versus* Pids

Sending a message to a non-existent name is an *error* …

… the assumption is that named processes should not disappear.

So, typically we name "static", long-lived processes.

A convention is that the named process gets the name of the module where it is defined … this assumes only one per module.



A hint of things to come

Suppose that we spawn the server process, and name it `server` …

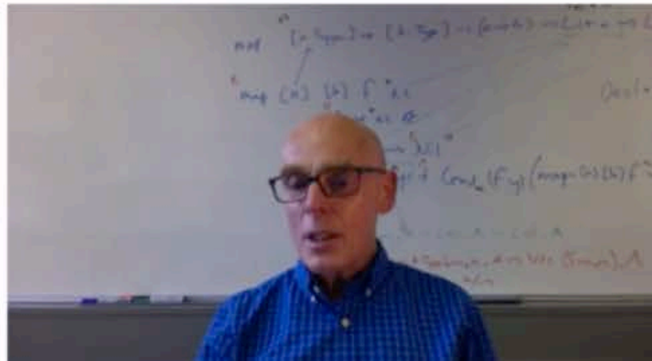
```
Server = spawn(M,F,[]),
register(server,Server),
```

… and other processes can then send messages to it: `server ! ping`.

```
server ! ping
```

What if the `ping` is sent before registration is complete?

This is a *race condition*.



University of Kent