Message passing: actors, mailboxes and the delivery assumption

Block 2. Actors, mailboxes and the delivery assumption

When a send never waits and every message goes to the mailbox of one named receiver, what does the model guarantee about order and delivery, and what does sharing nothing cost?

The plan of the session

Block 1	The RC 4000 nucleus (1969); shared memory and the scheduler; processes that share nothing; Kahn process networks; contested communications and three resolutions; the RC 4000's two resolutions; channels (CSP, occam, Go); buffer capacity.
Block 2	Actors and mailboxes (Hewitt, Agha, Erlang); fairness; the delivery assumption and the boundary with session 9; the cost of message passing, and channels against actors.
The lab	A message-passing library built by the students: a toy RC 4000 nucleus, one queue under two disciplines, the handshake, `select`, and a closing comparison.

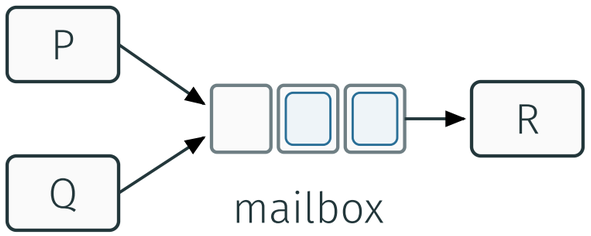
PART 7

Actors and mailboxes: asynchronous sends into a mailbox in arrival order

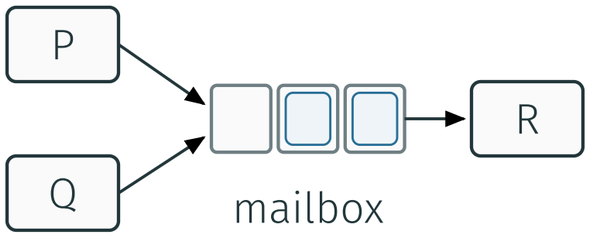
Hewitt 1973, Agha 1985, Erlang; the second family of message passing

The origins of arrival order in a queue

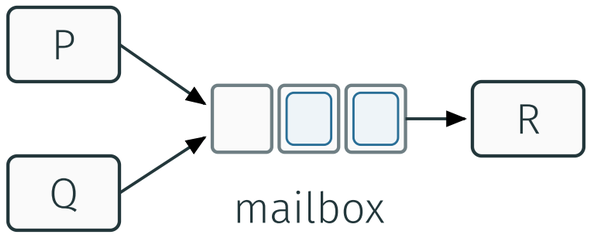
RC 4000, 1969
operating systems



Actors, 1973
artificial intelligence



Erlang, 1986
Ericsson; fault tolerance



Actors, 1973. The actor model began in artificial intelligence, not in operating systems: Hewitt, Bishop and Steiger 1973, out of Hewitt's earlier work on Planner; as far as is known, independently of the RC 4000.

Erlang, 1986. After the RC 4000 and the actors, Erlang reached the same answer at Ericsson, for another reason: fault tolerance.

Computation as a society of agents: “The model becomes a cooperating society of ‘little men’ each of whom can address others with whom it is acquainted” (page 236); “Every actor should act for himself or delegate the responsibility” (page 238).

Agha's actor primitives in OCaml

OCaml 5

toy_actor.ml ↗

```
type msg = Int of int | Ask of actor
and actor = { mutable behaviour : msg -> unit }

let mailroom : (actor * msg) Queue.t =
  Queue.create ()
let create b = { behaviour = b }
let send a m = Queue.push (a, m) mailroom
let become a b = a.behaviour <- b

let run () =
  while not (Queue.is_empty mailroom) do
    let (a, m) = Queue.pop mailroom in
    a.behaviour m
  done
```

`create` makes an actor from a behaviour and returns its address.

`send` sends asynchronously to an address and returns at once: the toy pushes the pair onto the mailroom and never waits.

`become` replaces the behaviour, so the next message is handled by the new one.

`run` takes one pending message at a time and calls its actor's behaviour on it: one turn, run to completion.

The counter actor and the toy's mailroom

OCaml 5

toy_actor.ml ↗

```
let rec counter n self = function
  | Int k -> become self (counter (n + k) self)
  | Ask reply -> send reply (Int n)
```

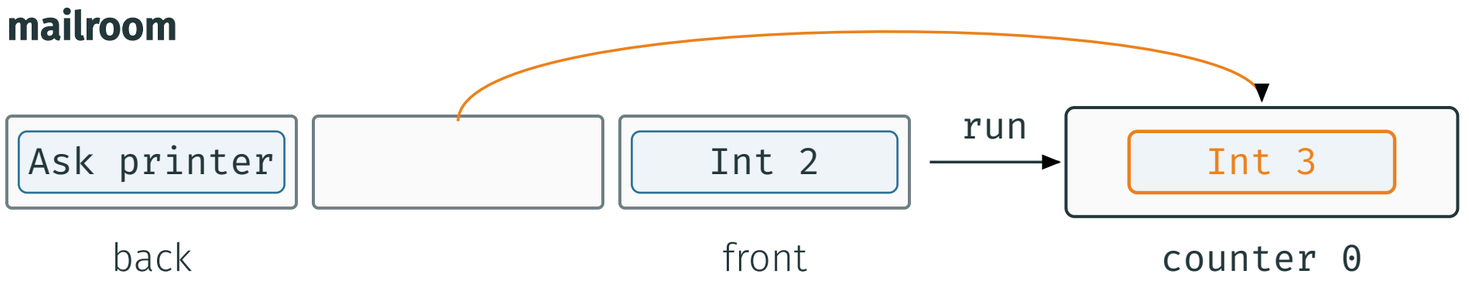
```
$ ocamlfind ocamlopt toy_actor.ml -o toy_actor
$ ./toy_actor
5
```

A send only enqueues. The sender continues at once; the message waits in the mailroom until `run` takes it.

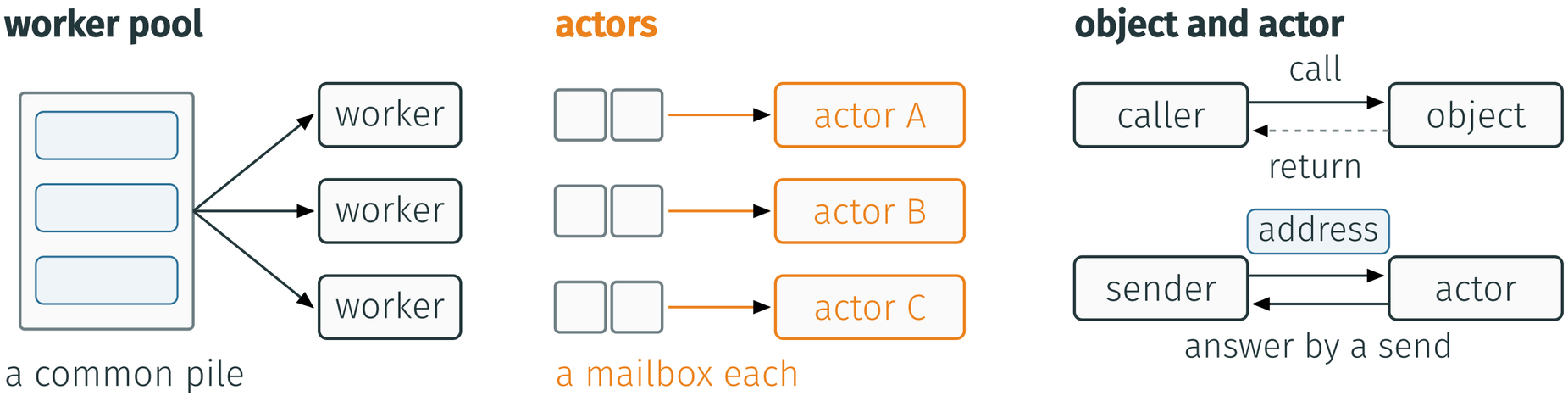
Each delivery runs to completion, the *Isolated Turn Principle* of De Koster et al. 2016: within one turn there is no low-level data race and no deadlock (pages 33 to 34).

The counter has no mutable variable of its own. Its state lives in the behaviour it becomes next.

Arrival order. The toy takes the front, `Queue.pop`: one order only. The model's order is arbitrary; a random pick among the pending messages models it.



Actors compared with worker pools, objects and Erlang's processes



Not a worker pool, interchangeable labour pulling work from a common pile: an actor is closer to a person with a mailbox of their own. In the kitchen, each cook has an order spike of their own, the ticket is pinned on *that* cook's spike, and only that cook reads it, in the order the tickets were pinned.

Classic actor and Erlang process. A classic actor is a handler that the runtime calls once per message; an Erlang process is a loop that blocks in `receive` (De Koster et al. 2016, pages 33, 35).

Not an object. An actor resembles an object that can be written to but never called: nothing returns, and a sender that needs an answer puts its own address in the message.

The guarantees of actor theory and their limits

Guaranteed

- every message sent is eventually delivered, after an arbitrary but finite delay (Agha 1985, page 23);
- a message arrives whole, an aim the model states rather than a guarantee it proves (page 20);
- one turn is isolated: no low-level race or deadlock inside it (De Koster et al. 2016, pages 33–34);
- a message is received only after it is sent (causality; Armstrong 2003, page 26);
- an actor sends to the actors it is acquainted with (Hewitt 1973, page 236, as a metaphor).

Not guaranteed

- arrival order, which is arbitrary (Agha 1985, page 22);
- a bound on the delay;
- a reply;
- that a delivered message is "meaningfully processed" (pages 23 to 24);
- freedom from races and deadlocks across several messages (De Koster et al. 2016, page 34).

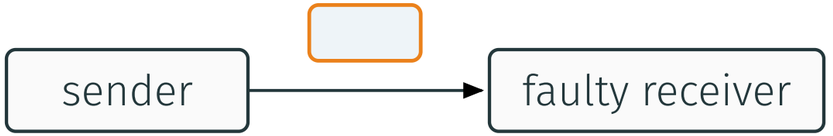
Agha's and Armstrong's arguments for asynchronous sends

no action at a distance



readiness learned by a send

fault isolation

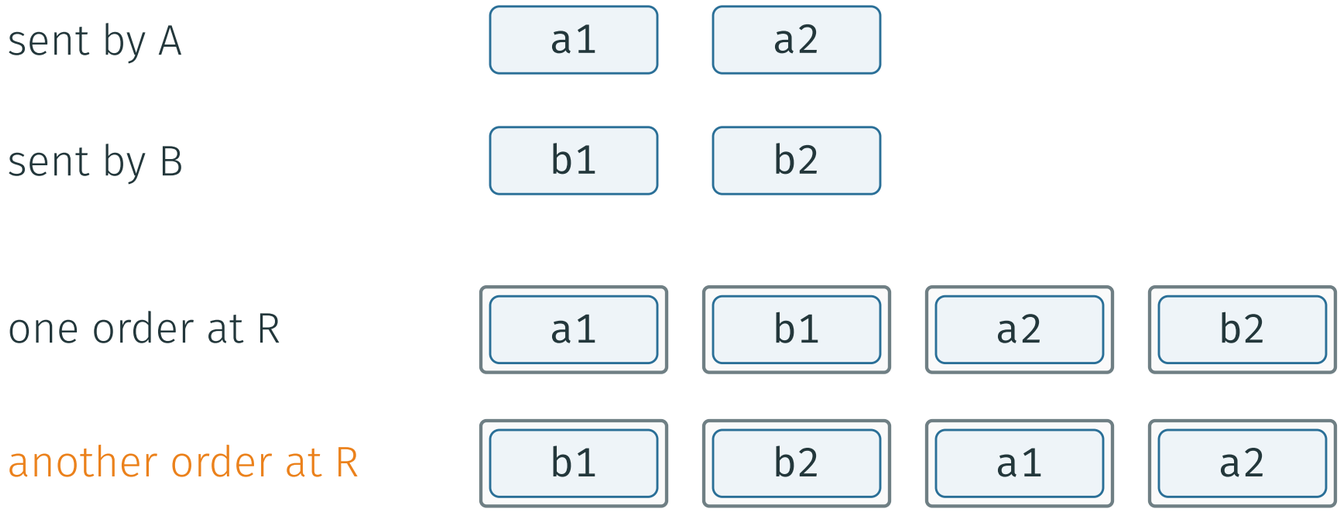


send and pray

Armstrong 2003, from fault isolation: “If process communication is synchronous then a software error in the receiver of a message could indefinitely block the sender of the message destroying the property of isolation” (page 23).

Delivery is then not guaranteed, across nodes or to a dead process; Armstrong calls the result “send and pray” (page 25).

Message order per pair of processes



A's two messages arrive in the order A sent them, and B's in the order B sent them.

Across different senders Erlang guarantees no order: how their messages interleave depends on the scheduling.

Mailbox semantics in Erlang

```
$ escript mailbox.erl
1. the send returned; the message waits in the mailbox
   received: a send returns at once
2. arrival order: [{a,1},
                   {a,2},
                   {a,3},
                   {a,4},
                   {a,5},
                   {b,1},
                   {b,2},
                   {b,3},
                   {b,4},
                   {b,5}]
   a's own order kept: true; b's own order kept: true
3. the selective receive took {urgent, 3} past 'first' and 'second'
4. is_process_alive(Dead) = false; sending anyway
   the send returned its message; nothing reports the loss
5. the monitor delivered {'DOWN', Ref, process, Pid, because}
```

1. A send returns before any process receives.

2. Each sender's own order is kept.

3. A selective `receive` takes a matching message from behind two earlier ones, so a mailbox is not a queue read from the front.

4. A send to the pid of a dead process returns normally, and nothing reports the loss.

The lost update across two messages

Erlang

counter.erl ↗

```
counter(N) ->
    receive
        {get, From} ->
            From ! {value, N}, counter(N);
        {set, M}    -> counter(M)
    end.

%% each of two clients:
C ! {get, self()},
receive {value, V} -> C ! {set, V + 1} end.
```

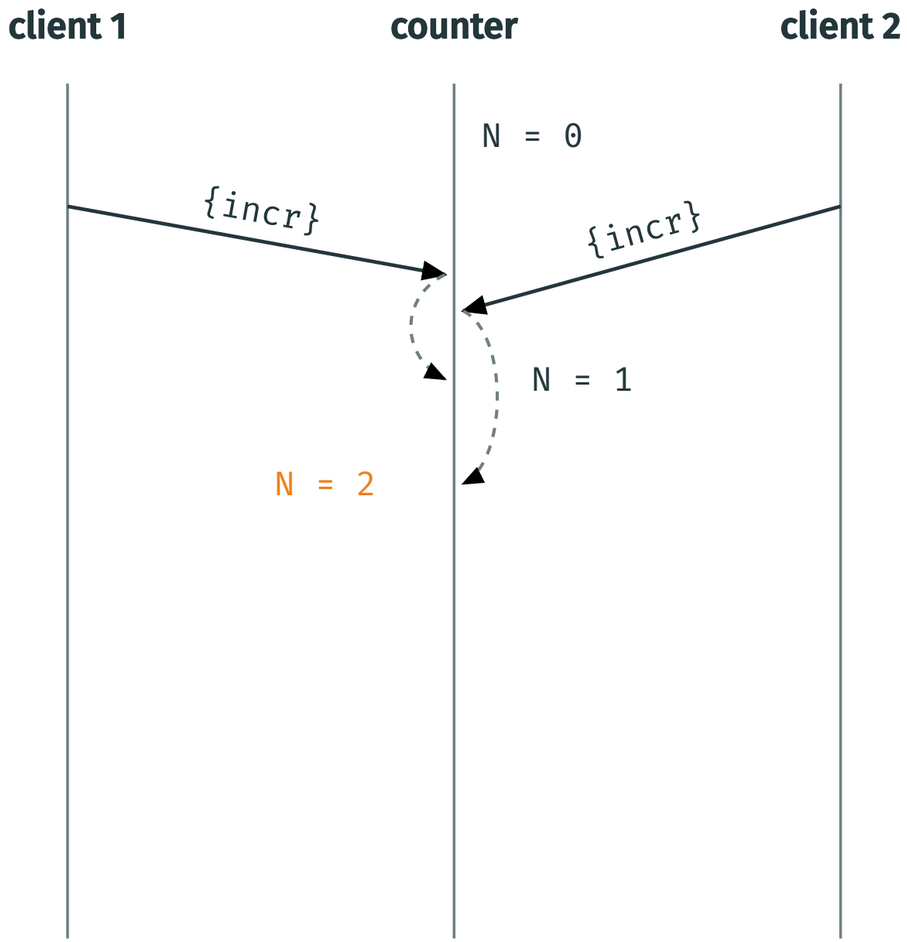
The lost update of part 2 returns without any shared variable: not as a data race, since no two accesses conflict, but as a race between messages.

A counter process owns its state, `N`.

Two clients each read it, then write it back plus one.

```
$ erlc counter.erl && erl -noshell -s counter start -s init stop
Erlang/OTP 29, ERTS 17.1, schedulers 12
run(3000): final value 1 (lost): 2953, final value 2: 47
```

The lost update and its repair, message by message



Both clients send get; the two requests wait in the counter's mailbox in arrival order, client 1's first in this run.

The counter takes client 1's get and answers 0.

It takes client 2's get and answers 0 too: neither client has written yet.

Both clients send set 1, the value read plus one.

The counter takes the first set: N = 1.

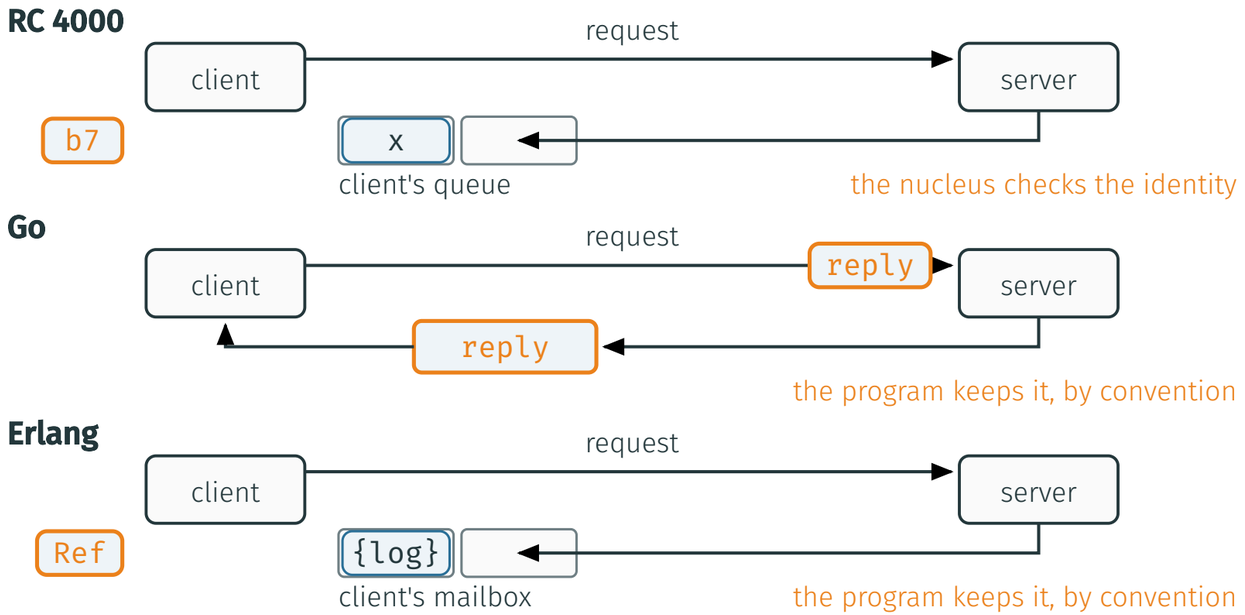
It takes the second set: N = 1 again, after two increments.

The repair makes the increment one message: each client sends incr.

The counter's own sequential code applies both: N = 1, then 2.

> The counter's own code is sequential; the race lies in the order in which its mailbox merges the two clients' messages.

A request and its reply in the RC 4000, Go and Erlang



RC 4000: the request travels in buffer b7.

Go: the request carries a fresh channel, reply.

Erlang: the request carries a fresh reference, Ref.

The client takes its reply past the other message, by its identity, and who checks that identity: in the RC 4000 the nucleus; in Go the program, by convention; in Erlang the client's own pattern.

The answer comes back in the same buffer, into the client's queue behind x; wait answer(b7) takes it by its identity.

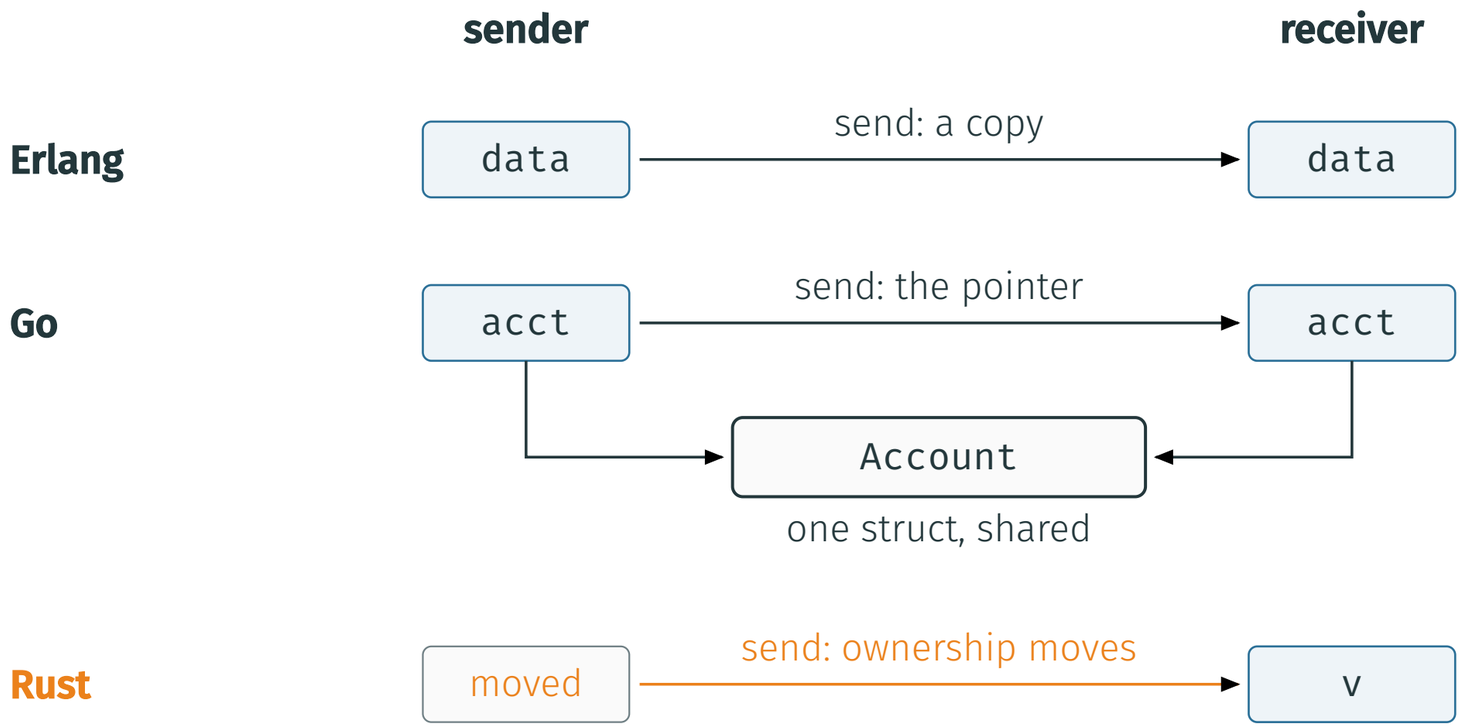
The server sends the answer on that channel, which, by convention, only the client reads.

The answer comes back into the client's mailbox beside other messages, tagged with Ref, and the receive's pattern picks it out.

Message order in the systems of the story

Model	Order
Go channel	a first-in-first-out queue: with one sender and one receiver, values are received in the order sent; no order among concurrent senders (the Go specification)
CSP 1978	no queue; each exchange is one event between two sequential processes, so the exchanges of a pair are ordered; choice among senders arbitrary (Hoare 1978)
Actor theory	arbitrary (Agha 1985)
Erlang	per pair of processes (Armstrong 2003), for ordinary messages and the default mailbox; with priority messages enabled (OTP 28.0 and later), two messages of one sender can stand in the queue out of their sending order; selective receive can take a later message first (the Erlang reference manual)
RC 4000	one queue per process; two primitives allow any order (Brinch Hansen 1970)

The data carried by a message



Erlang copies the data of a message, except reference-counted binaries and literals, which are shared on one node (the efficiency guide, "Processes").

Rust moves ownership with the send for a value that is not `Copy`; a `Copy` value is copied, and an `Arc` moves a handle to data that stays shared (the Rust book, 16.2 and 16.3; `std::sync::Arc`).

A Go channel passes the value it is given, so a sent pointer stays shared.

Summary of part 7

The actor family, as the section has settled it:

The send	returns at once, into the mailbox of one named receiver
The order of arrival	left arbitrary by the model; fixed per pair of processes by Erlang, for ordinary messages
The delivery	assumed by the model
The race	a race between messages, in place of the data race

> Left to the next section: fairness, of which the assumed delivery is one form, and how the two families state it.

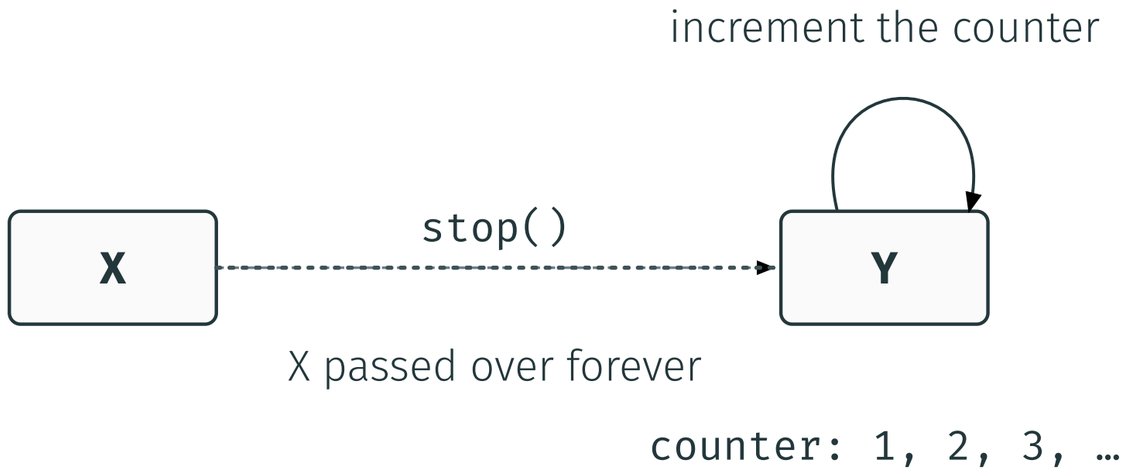
PART 8

Fairness: eventual choice and eventual delivery

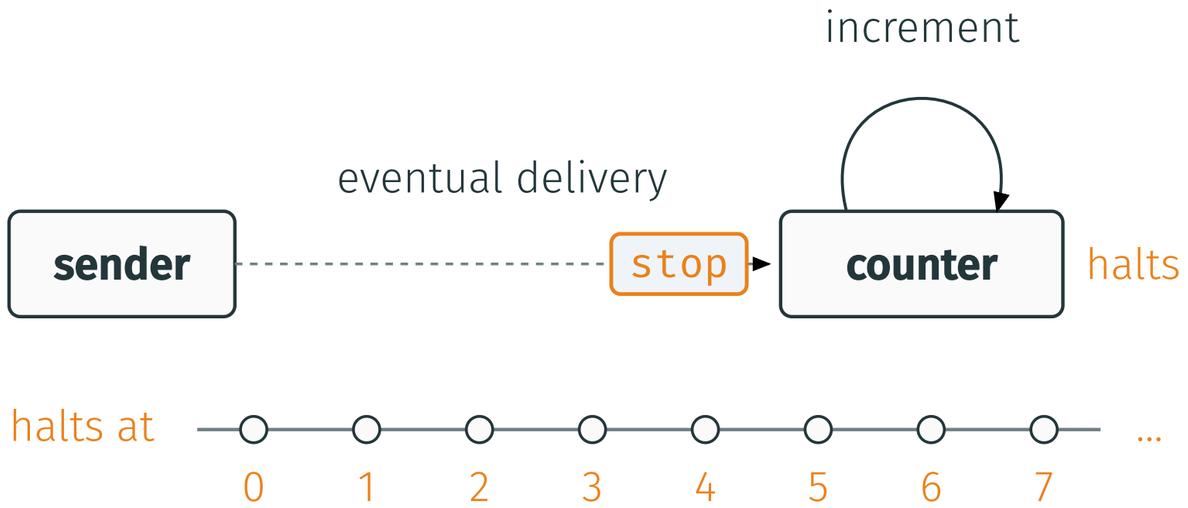
CSP (1978), actor theory and Erlang's manual

Fairness in CSP and in actor theory

CSP: a choice between offers



Actor theory: eventual delivery



Hoare's example. X sends `stop()` while Y loops, either receiving the stop or incrementing a counter.

Hoare's refusal. The language does not forbid that, because the implementation would then have to complete programs with unbounded nondeterminism; so "a proof of correctness must not rely on this property" (Hoare, 1978, page 676).

Unbounded nondeterminism. The stop is guaranteed to arrive, so the counter always halts, but at any number at all: a program certain to terminate, with infinitely many possible results.

An unfair implementation. An implementation that always prefers the counter passes X over forever.

Clinger's example. In actor theory, the guarantee of eventual delivery is a fairness assumption: a counter increments until a "stop" message arrives.

Delivery and loss in CSP, actor theory and Erlang

	CSP, Hoare 1978	Actor theory, Agha 1985	Erlang, to a live process on the same node (the manual)	Erlang, to a dead process or another node (Armstrong 2003)
A ready message is eventually taken or delivered	not guaranteed	guaranteed	arrives after a positive time, unless the receiver has terminated or the alias is inactive	not guaranteed
A message can be lost	no such thing exists	no, by assumption	only one sent through an inactive alias, dropped before the queue	yes

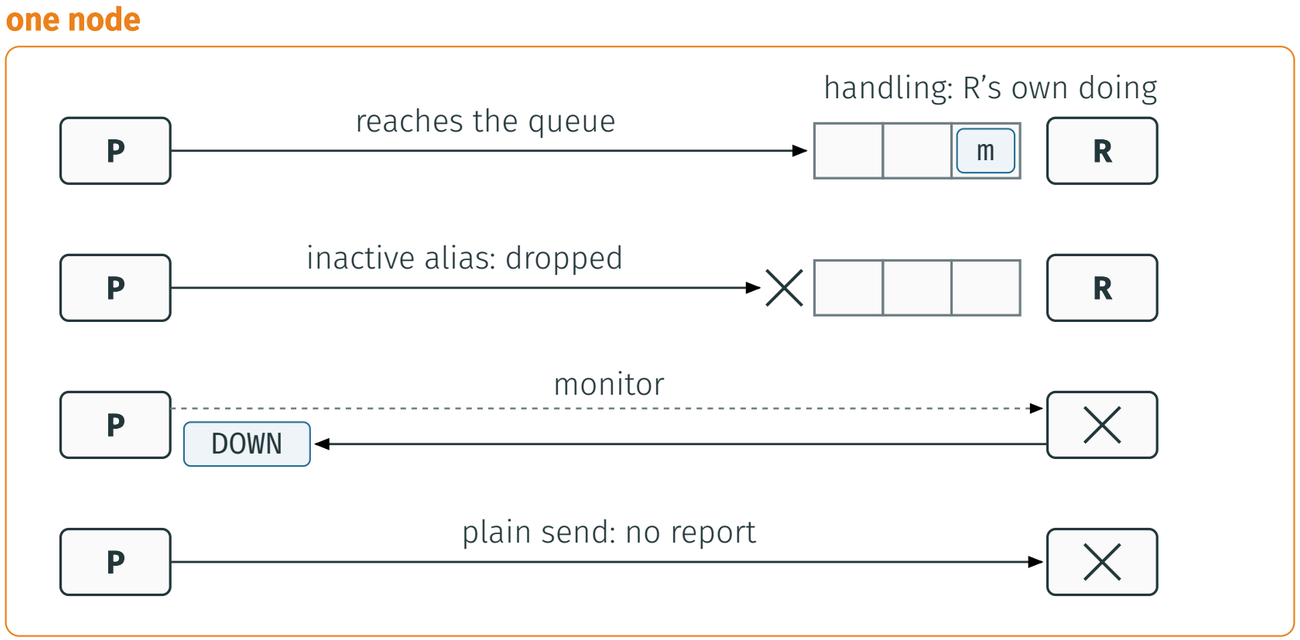
> The last column of the table, a dead process or another node, is where delivery itself can fail.

PART 9

The delivery assumption: the boundary with session 9

Delivery on one node, a death reported where requested, and what session 9 changes

The delivery assumption



Arrival. Every message sent to a live process reaches its queue.

Handling. Whether the process handles the message is its own doing.

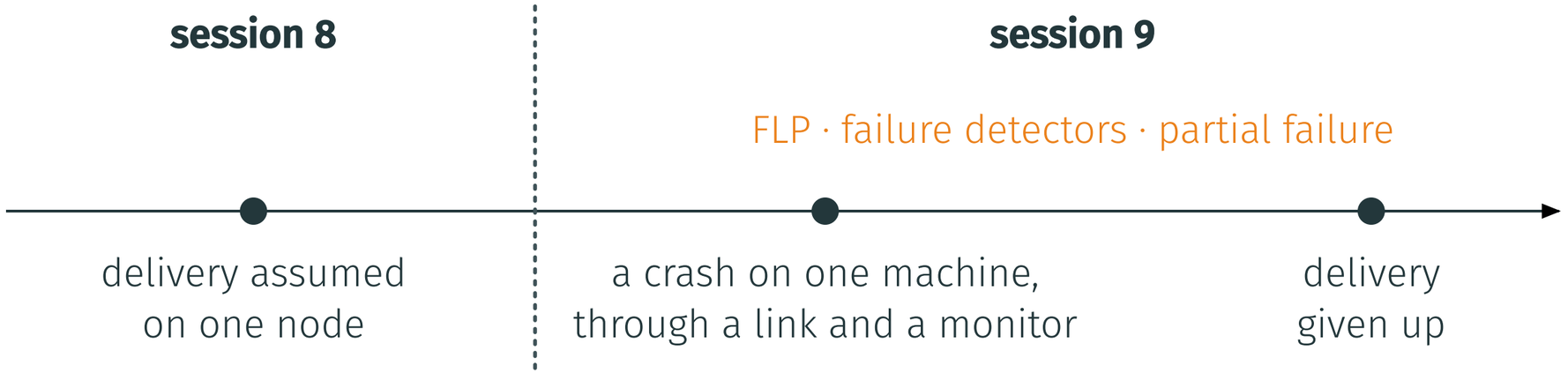
A plain send. A plain send to a dead process reports nothing.

The one exception. A message sent through an inactive alias is dropped before the queue.

A death reported where requested. The runtime reports a death exactly where the program requests the report: an Erlang monitor turns it into a `DOWN` message (the module erlang, `monitor/2`); the RC 4000's nucleus returned a dummy answer (Brinch Hansen, 1970, pages 239 to 240).

A teaching assumption. The RC 4000's paper states it for a live receiver on one machine (Brinch Hansen, 1970); neither Erlang's manual nor Go's specification states it as such.

The boundary with session 9



The start of session 9. A crash on one machine, observed through a link and a monitor.

The subjects of session 9. The impossibility result of Fischer, Lynch and Paterson (1985), failure detectors and Waldo's partial failure (1994) belong there.

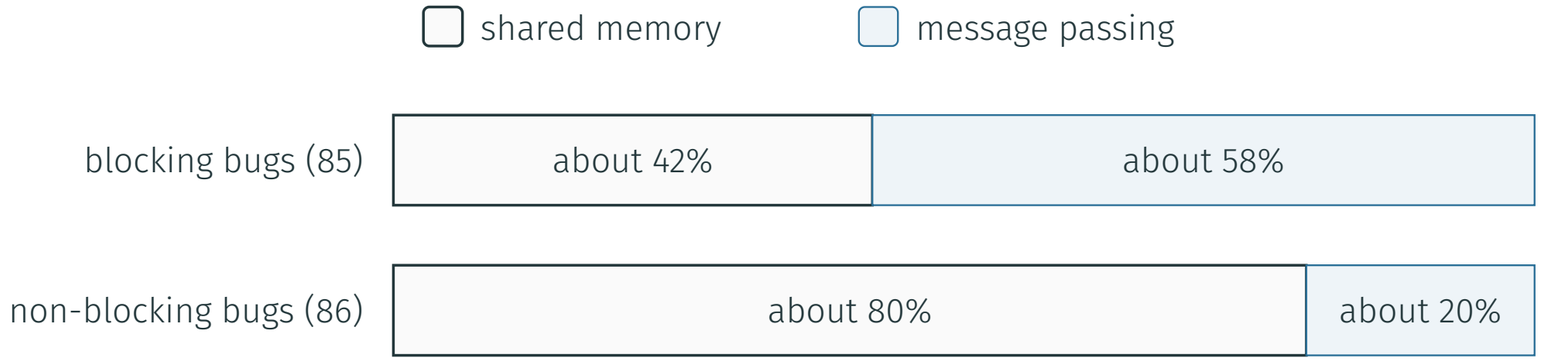
Delivery given up. Only then does session 9 give up delivery.

PART 10

The cost of message passing, and the relation between channels and actors

the bug study of six Go systems; sharing let back in; the π-calculus and its two limits; channels against actors

The bug study of Tu, Liu, Song and Zhang



"Contrary to the common belief that message passing is less error-prone", more blocking bugs came from message passing (page 6).

Read together, in the systems studied, message-passing bugs were mostly blocking and shared-memory bugs mostly non-blocking.

The study's other half points the other way: far fewer non-blocking bugs came from message passing (pages 10 to 11).

Consistency with the story	The price of removing sharing is waiting.
Uses of each primitive (§3.2)	Shared-memory primitives are used more often, so by the study's count message passing is even likelier to cause a blocking bug.
Open question	Whether message passing creates or prevents bugs overall.

Sharing through a sent pointer

Go

shared_pointer.go ↗

```
type Account struct{ Balance int }

ch := make(chan *Account)
go func() {
    acct := <-ch
    fmt.Println("receiver reads the balance:",
        acct.Balance)
}()
acct := &Account{Balance: 100}
ch <- acct          // the pointer is sent
acct.Balance = 200 // a write after the send
```

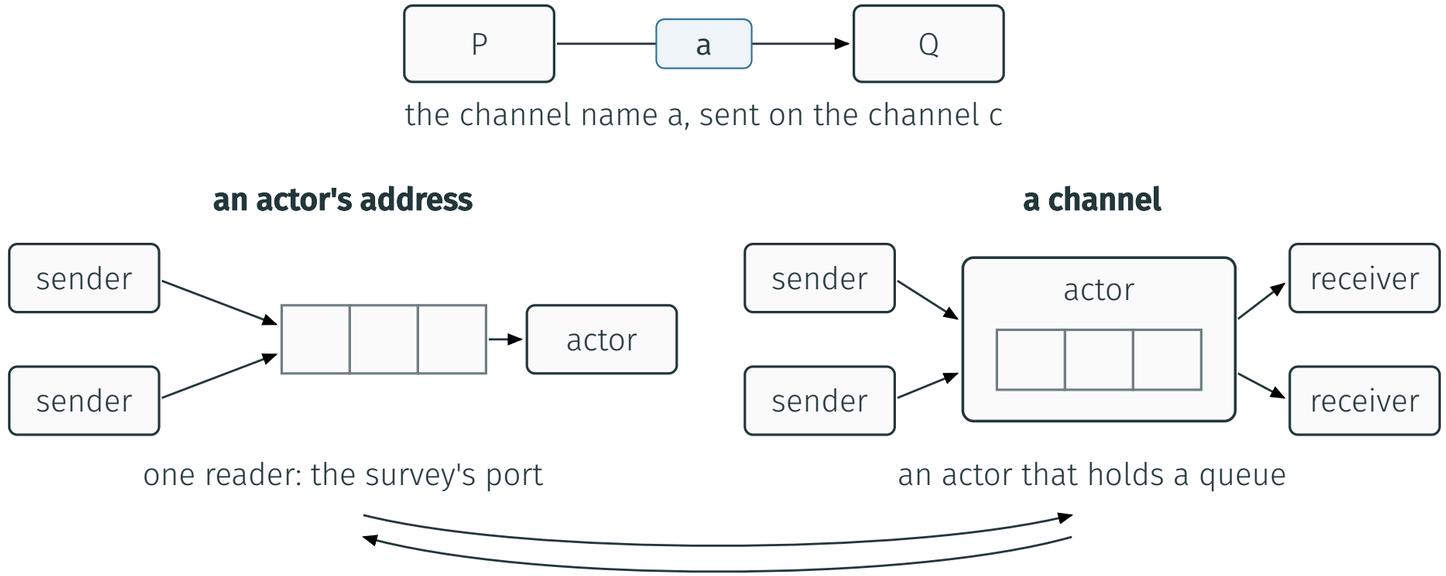
A pointer sent on a channel stays shared.

"Share by communicating" is a convention, and a program can break it without the compiler detecting it.

The race detector, in an instrumented run, reports the write after the send.

```
$ go run -race shared_pointer.go
receiver reads the balance: 100
==================
WARNING: DATA RACE
Write at 0x00c000116048 by main goroutine:
  main.main()
Previous read at 0x00c000116048 by goroutine 7:
  main.main.func1()
Goroutine 7 (finished) created at:
  main.main()
==================
sender's final balance: 200
Found 1 data race(s)
```

Mutual encodings of channels and actors and their limits



An actor's address is a channel with exactly one reader, the survey's port (Andrews and Schneider 1983, page 26).

In the calculi studied, each family can be translated into the other (asynchronous channels without choice, in Fowler, Lindley and Wadler 2017), with what a translation preserves depending on it.

No fairness assumption is carried across by the translations studied: Go's channels guarantee no eventual delivery, actor theory assumes it (Agha 1985), so fairness has to be assumed on each side separately.

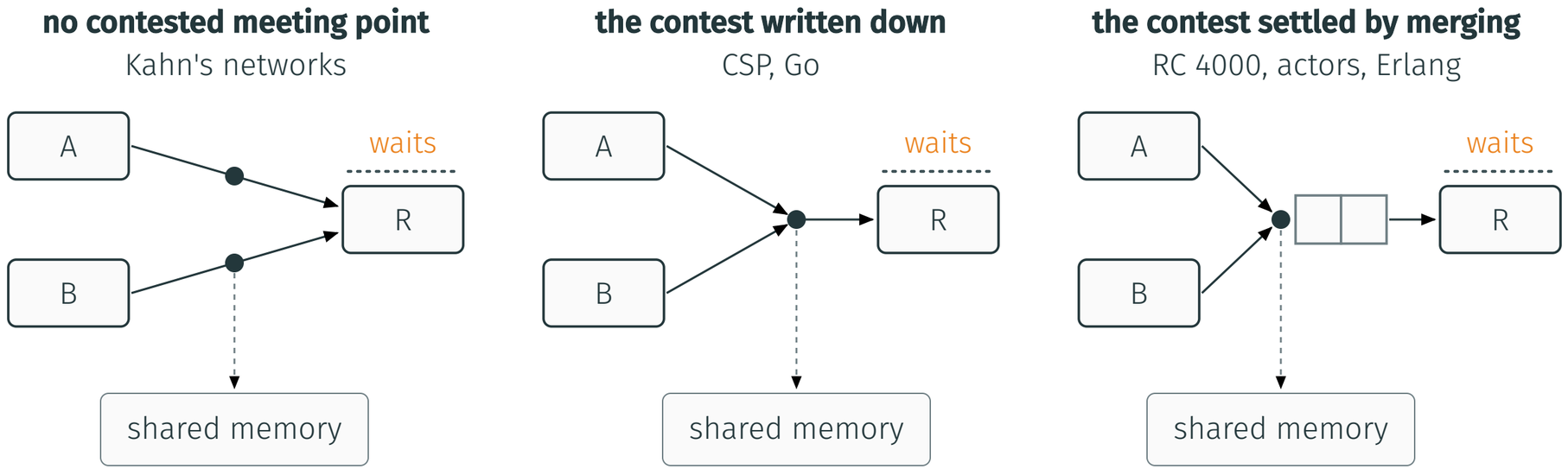
A channel can in turn be written as an actor that holds a queue.

Mixed choice, a choice that waits on a send and a receive at once, has no uniform encoding in the asynchronous π-calculus (Palamidessi 1997); a choice among inputs only can be encoded (Nestmann and Pierce 1996), and Go's `select` offers mixed choice.

Actors and channels compared

	Channel (Go)	**Actor (Erlang)**
A send addresses	a conduit, independent of processes	a specific actor
Who reads	any process holding it	only the owner
Several sources	the receiver chooses (`select`)	one mailbox; patterns select messages
Resolution of a contested communication	a guarded choice in the program; a merge left to the scheduler	arrival order in the mailbox
Buffer	often bounded (backpressure)	conceptually unbounded
Failure	not in the model (Go)	usually built in (Erlang)
Across machines	usually local	addresses often work across nodes
The RC 4000 (1969)	its replies: identified, not contested	its requests: one queue, served in arrival order

Summary of block 2



one trusted place in each runtime, on one node

This holds as long as processes share nothing by any other route, messages carry values, the runtime delivers them as part 9 assumes, and no process is nondeterminate inside itself.

CSP and Go let the programmer write the contest down.

On one node, in every runtime of this story, and in any runtime whose processes share one address space, the shared memory is still there, hidden in one trusted place.

Kahn's networks show what happens if no meeting point is ever contested.

The RC 4000, actors and Erlang let the system settle it by merging.

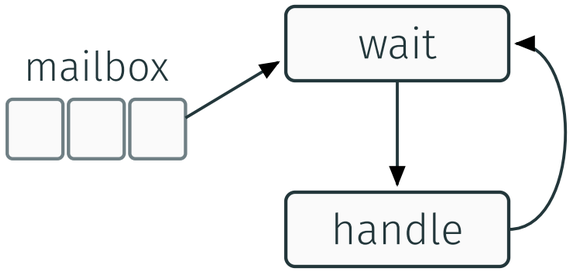
The cost is paid in waiting, and in races between messages, which the protocol must rule out.

Details of block 2

supporting slides, each reached by the appendix button of a slide of the lecture, each with a button back

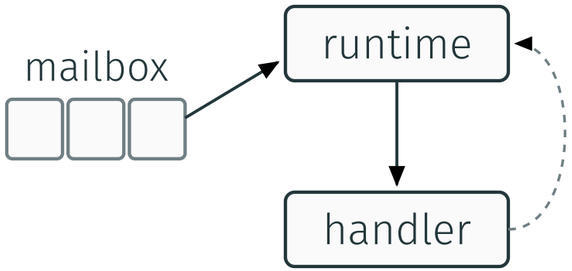
The RC 4000 process, the classic actor and the Erlang process

RC 4000 process



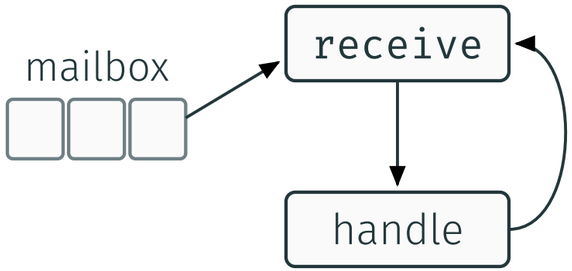
wait, handle, wait again

classic actor



become: next behaviour

Erlang process



`loop(State)`

A classic actor inverts that structure: no loop, never waits; a handler that the runtime calls once per message, and `become` records the behaviour for the next message. Classic actors are therefore event handlers, a state machine written by hand, which is why programming with them resembles asynchronous programming.

An Erlang process restores the loop: it loops, blocks in `receive`, and passes its state through recursion. That is the split between Classic Actors and the Processes family (De Koster et al. 2016, pages 33, 35).

Delivery in TCP and in the actor model

	TCP	**actor model**
order	kept per connection	none guaranteed
delivery	obtained by retransmission	assumed

> "Delivery guaranteed" is an idealization of the model, not a TCP-like protocol. Agha defends it the way a semantics defends an infinite stack: unbounded buffers do not exist physically, but the model abstracts from such limits (Agha 1985, page 23).

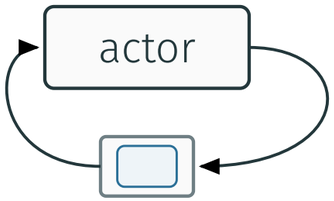
Agha's argument for asynchronous communication

no action at a distance



readiness learned by a send

a send to itself



needs a buffer

a fast sender

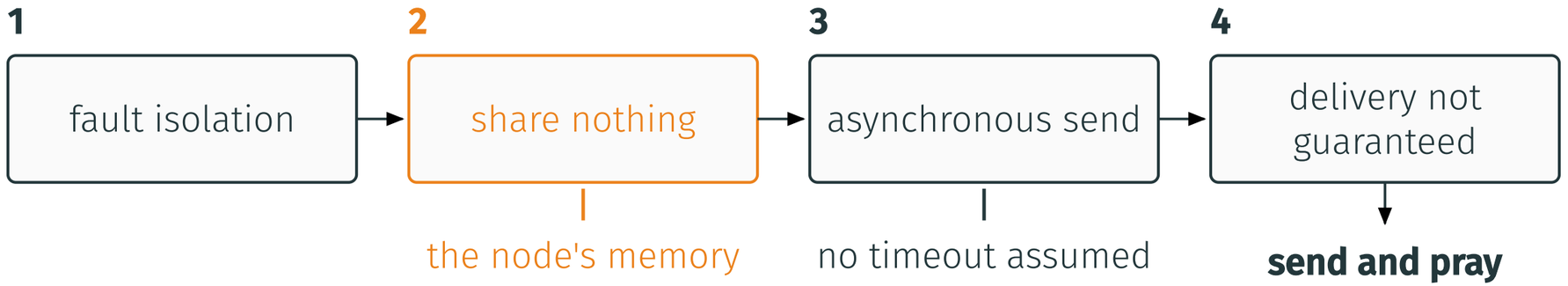


not held to one character

Without buffering, an actor cannot send a message to itself, and recursive computation requires that.

A fast sender, a typist faster than an editor, should not be held to one character at a time.

Armstrong's argument from fault isolation and its gaps



1. The essential problem in a fault-tolerant system is fault isolation: errors in a faulty module must not reach a correct one (page 17).

2. Any shared resource lets an error in one process corrupt it, so processes share nothing (pages 19, 23).

3. Message passing must be asynchronous: “If process communication is synchronous then a software error in the receiver of a message could indefinitely block the sender of the message destroying the property of isolation” (page 23).

4. Delivery is not guaranteed, across nodes or to a dead process; the sender can learn that a message arrived only from a confirmation sent back (pages 22–23).

Armstrong calls the result “send and pray” (page 25).

No timeout. Step 3 assumes a synchronous send with no timeout, which a timed send would bound.

One shared resource left. An unbounded mailbox leaves the node's memory, which a stuck receiver's mailbox can exhaust for every process. Armstrong concedes a version of this himself, for operating-system processes that share one CPU and one memory, calling isolation “only true to a first approximation” (page 17).

A send to a dead partner, in Erlang and on the RC 4000

Send	Outcome
Erlang, to the pid of a dead process	returns normally; nothing reports the loss
Erlang, to a name no longer registered	raises `badarg` (`registered_send.erl`, run on record, OTP 29)
RC 4000	a dummy answer
Erlang, with a monitor	the partner's death becomes an ordinary message in the mailbox, `{'DOWN', Ref, process, Pid, Reason}` (the Erlang reference manual)

The reply matched by a reference

```erlang
Ref = make_ref(),
Server ! {request, self(), Ref, Data},
receive
    {reply, Ref, Result} -> Result
after 1000 -> timeout
end
```

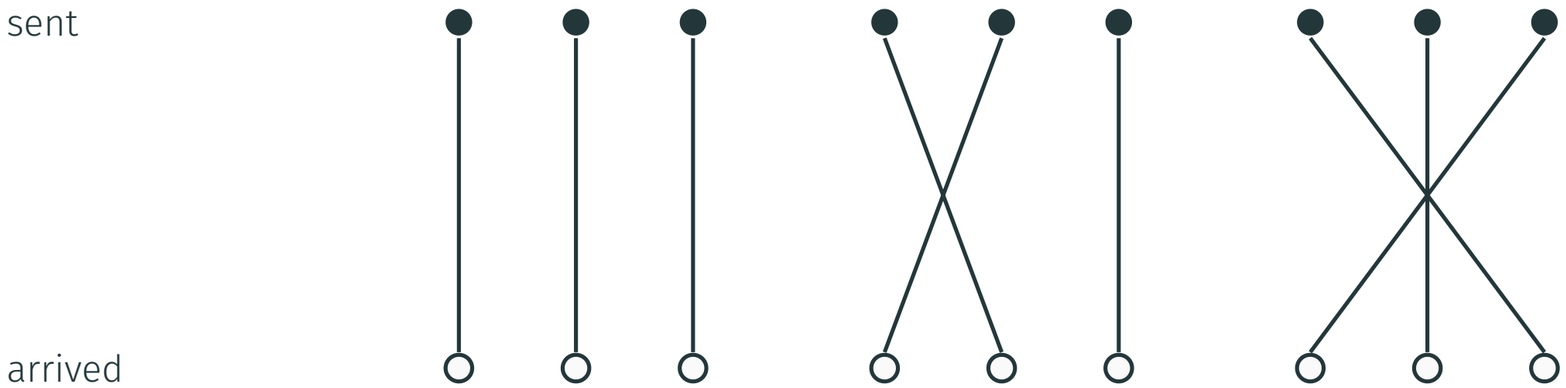
The mailbox that merges requests also receives replies.

The request carries a unique reference, `Ref`, in place of the RC 4000's buffer identity.

The reply comes back into the same mailbox as every other message; the selective `receive` finds it by `Ref`.

The program's pattern checks the identity, where on the RC 4000 the nucleus checked it.

Sending order, arrival order and receiving order



the actor model: any arrival order

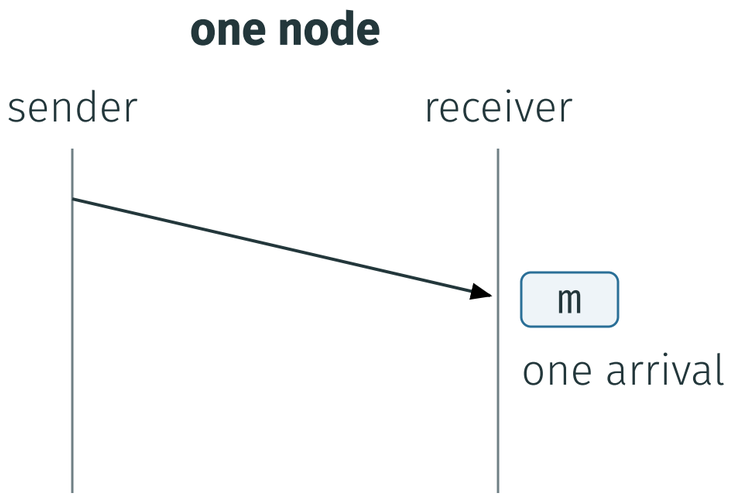
Erlang keeps the order between one sender and one receiver for ordinary messages: they arrive in the order sent, so the lines run straight.

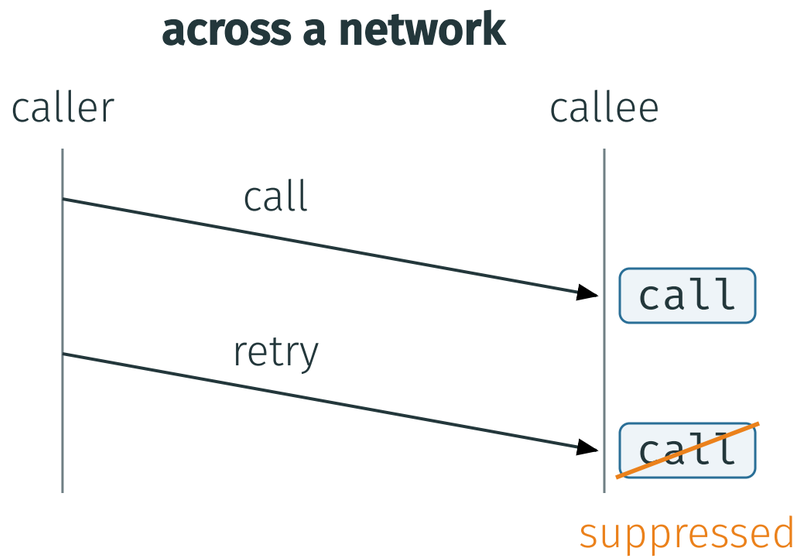
A Go channel with one sender and one receiver: the order sent is the order received, and nothing crosses.

The selective receive takes {urgent, 3} first, past the two earlier messages, as the recorded run of mailbox.erl shows: the orange line is the message taken out of order, the other two shift right, and the crossing is the pattern's doing, not the transport's.

The actor model leaves arrival order arbitrary: any of the six orders may arrive, three drawn here.

Duplicates across a network



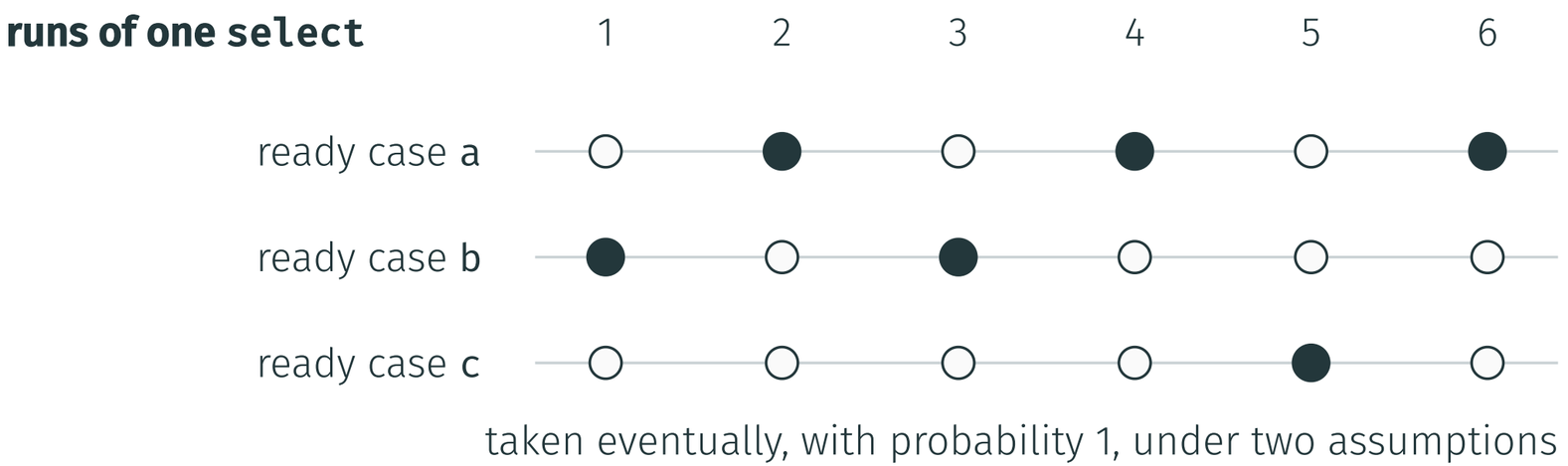


Duplicates arrive with retries across a network.

After an exception, the call had run "either once or not at all—the user is not told which" (Birrell and Nelson 1984, p. 49).

The remote procedure call protocol of Birrell and Nelson (1984) suppressed them.

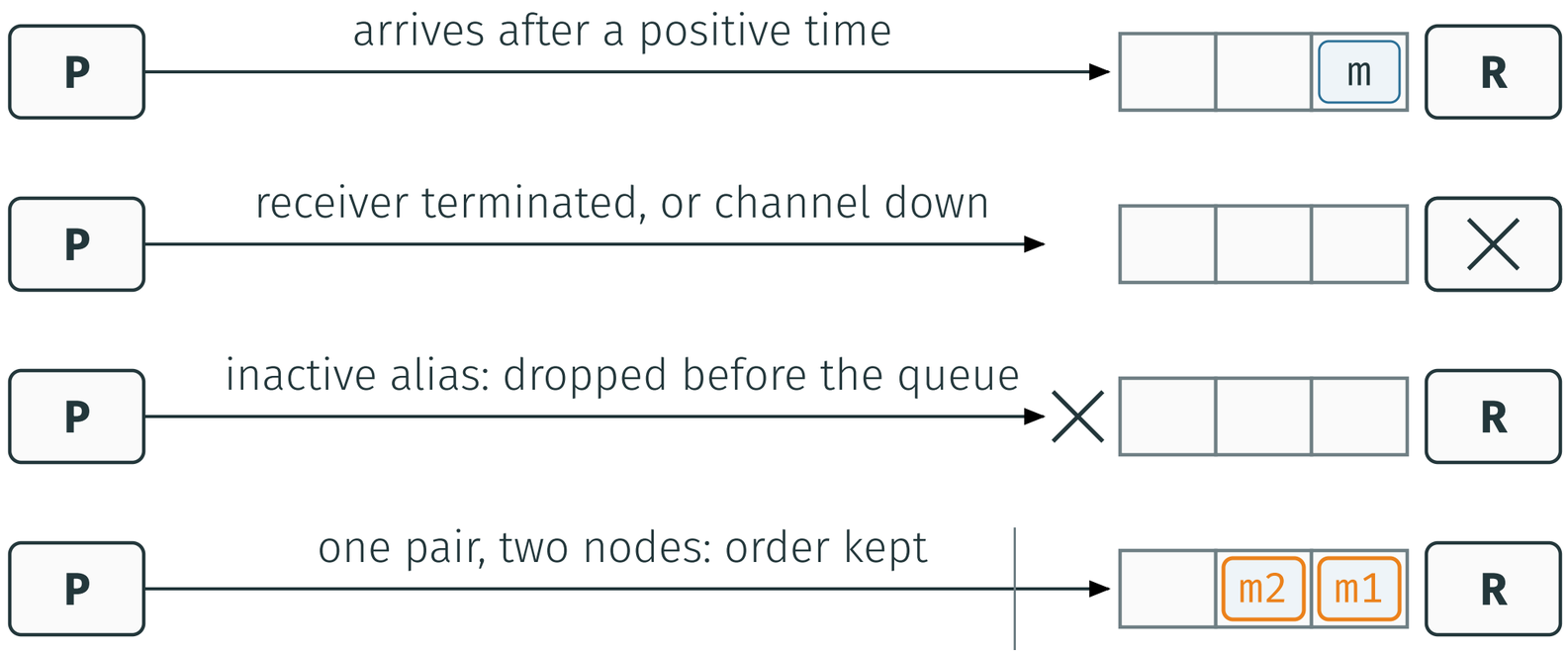
Go's `select` and eventual service



An inference, with probability 1. A case that stays ready is taken eventually, if the choices are independent and the goroutine runs the `select` again and again while the case stays ready.

Not stated by the specification. Eventual service of a ready case, and any order among blocked senders on one channel.

Arrival in Erlang's manual



The named exceptions. A terminated receiver; a distribution channel that goes down.

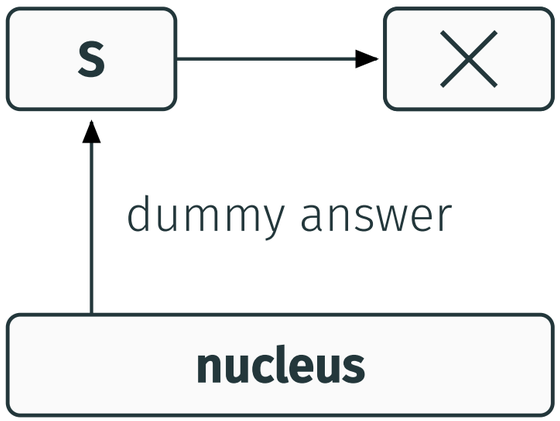
Arrival without handling. Arrival says nothing about the process handling the message.

The inactive alias. A message sent through an inactive alias is dropped before it enters the queue.

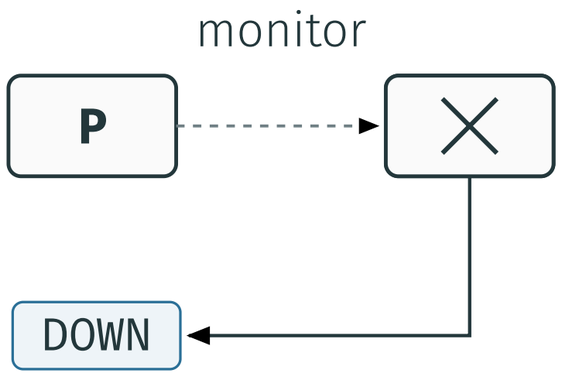
Order per pair. The order between one pair of processes is kept, even across nodes.

A death reported on one runtime

RC 4000 nucleus



Erlang monitor



plain send in Erlang



no report

The RC 4000. The nucleus returned a dummy answer (Brinch Hansen, 1970, pages 239 to 240).

A plain send. A plain send to a dead process reports nothing.

Erlang. A monitor turns a death into a `DOWN` message (the module `erlang`, `monitor/2`).

Sharing after a send in safe Rust

Safe Rust closes the data-race hole, not the sharing (*The Rust Programming Language*, chapter 16; `std::sync::Arc`).

A send	moves ownership
What stays shared	an `Arc`, mutated only through a type that synchronizes its own access: a `Mutex` an `RwLock` an atomic a `OnceLock`
Safe code	the type system forbids unsynchronized conflicting accesses, not shared state
Unsound `unsafe` code	reopens the hole

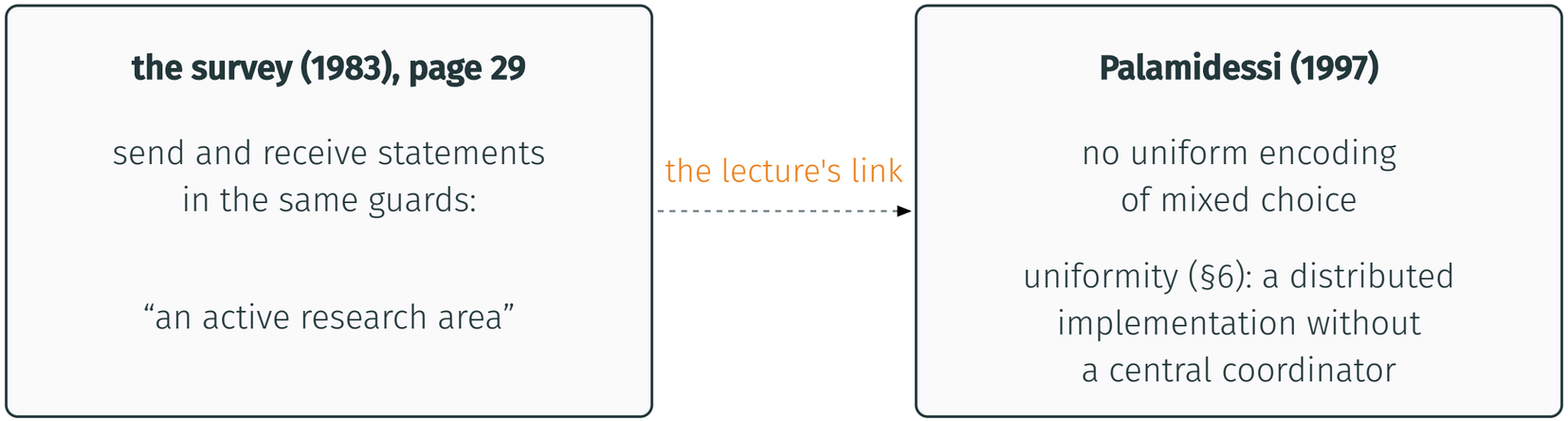
Limits of the mutual encodings of channels and actors

> **Mixed choice**: a choice that waits on a send and a receive at once.

	Mixed choice	**Fairness**
The calculi studied	no uniform encoding in the asynchronous π-calculus that preserves a reasonable semantics (Palamidessi 1997); a choice among inputs only can be encoded (Nestmann and Pierce 1996)	no fairness assumption carried across by the translations studied
Go	`select` offers it	channels guarantee no eventual delivery
Actors	pure actors cannot express it directly	eventual delivery assumed (Agha 1985, page 23)

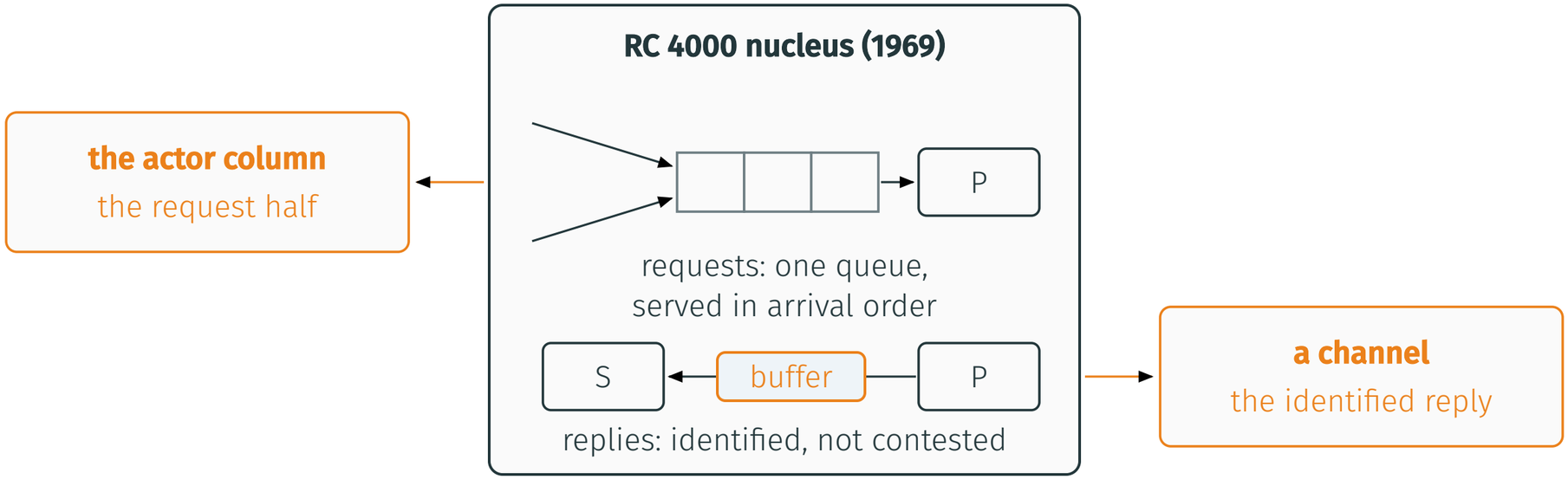
> Fairness has to be assumed on each side separately.

Mixed choice in the survey of 1983



Palamidessi's uniformity criterion models a distributed implementation without a central coordinator (§6); reading the survey's problem as that requirement is the lecture's own link.

The RC 4000 compared with actors and channels



Its replies are identified conversations that no process contests, the idea that Go's reply channel and Erlang's reference both reuse.

The lecture's two blocks are the identified reply taken to its full form as a channel, and the request half taken to its full form.

One system used two of the three resolutions by default; its two further primitives also let a program choose.

Earlier accounts of the lecture's argument

Lee (2006), *The Problem with Threads*	threads "wildly nondeterministic" (page 5); Kahn networks deterministic by their blocking reads (page 11) nondeterminism "explicit in programs" (page 15), for example an explicit "NondeterministicMerge" (page 11)
Lee and Parks (1995), *Dataflow Process Networks*	§2.2: the ways nondeterminism enters a Kahn network, the table in the appendix of block 1
Van Roy and Haridi (2004), *CTM*, **chapters 4 to 5**	deterministic dataflow first, then ports, since in client/server "the order of the requests cannot be determined in advance" (UCLouvain's LINFO1131, lecture 5)
Fowler, Lindley and Wadler (2017)	typed translations between asynchronous channels without choice and actors; input-guarded choice added, its emulation by actors left open
The lecture's addition	the RC 4000 the resolution of a contest as the comparison's axis