XCHAN

"XCHANs: Notes on a New Channel Type"

CPA-2012 at Abertay University
Dundee, Scotland

Øyvind Teig

Autronica Fire and Security

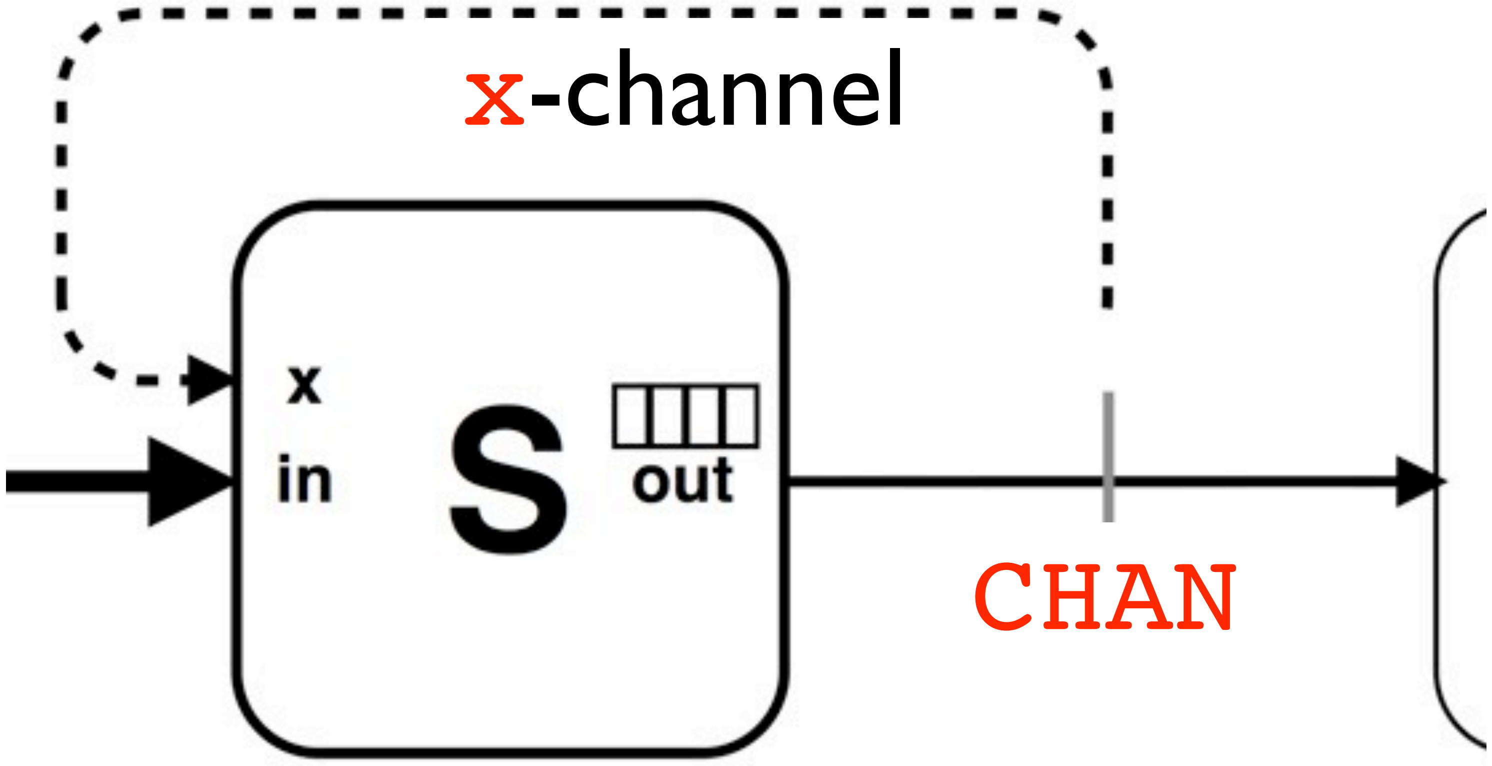
Trondheim, Norway

http://www.teigfam.net/oyvind/pub/pub_details.html#XCHAN

http://wotug.org/paperdb/

Version 28. Aug.2012 07:29 (55 incremental pages)

XCHAN = x-channel + CHAN



x-channel

CHAN

Background

- From discussions at Autronica
- Not implemented
- Goal for me was to try to merge asynchronous and synchronous "camps"..
- ..to arrive at a common methodology
- To make it "easier" to comply to SIL (Safety Integrity Level) approving according to IEC 61508 standard for safety critical systems
- Assumed implementation *loosely* based on implemented ideas with `EGGTIMER` and `REPTIMER`. ([9] CPA-2009 paper)

```
XCHAN          OF BYTE my_xchan:
```

asynchronous, nonbuffered

asynchronous, buffered

```
XCHAN (100) OF BYTE my_xchan:
```

```
XCHAN (...) OF BYTE my_xchan:
```

Sender is notified as to its *success* or "*failure*"

```
XCHAN (...) OF BYTE my_xchan:
```

Sender is notified as to its *success* on return of send:
- data moved to buffer
- data moved to receiver

```
XCHAN (...) OF BYTE my_xchan:
```

Sender is notified as to its "*failure*" on return of send:

- buffer full
- receiver not present

```
XCHAN (...) OF BYTE my_xchan:
```

Sender is notified as to its "*failure*" on return of send:

- buffer full
- receiver not present

It *always* `returns`!

If "*failed*" to send on `XCHAN`:

If "*failed*" to send on `XCHAN`:

"Not sent" is *no fault*!

If "*failed*" to send on `XCHAN`:

"Not sent" is *no fault*!

But a contract to send later

If not sent on `XCHAN`:

- listen to `x`-channel (in an `ALT` or `select`)
- resend old or fresher value when it arrives
- this send will always succeed

If not sent:

"channel-ready-channel"

- listen to `x`-channel (in an `ALT` or `select`)
- resend old or fresher value when it arrives
- this send will always succeed

If not sent:

- listen to `x`-channel (in an `ALT` or `select`)
- resend old or fresher value when it arrives
- this send will always succeed

This contract (design pattern) between sender and receiver must be adhered to

All said!

The rest is really rationale and

Tradition

Send and *forget*

New

Send, if not sent then *don't forget* `x`-channel

Send and *forget*

Asynchronous

Send, if not sent then *don't forget* `x`-channel

Asynchronous

Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

Send, if not sent then *don't forget* x-channel

Asynchronous

Full flow control (bridge only ever full)

Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

..finding "enough" buffer size..

Send, if not sent then *don't forget* x-channel

Asynchronous

Full flow control (bridge only ever full)

Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

Forget <u>means</u> no application handling

Send, if not sent then *don't forget* x-channel

Asynchronous

Full flow control (bridge only ever full)

Full application handling (but don't *forget* x-channel)

Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

Forget means no application handling

Those programmers..

Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

Forget means no application handling

Those programmers..

..could love this..

Send, if not sent then *don't forget* x-channel

Asynchronous

Full flow control (bridge only ever full)

Full application handling (but don't *forget* x-channel)

Send and *forget*

Restart if b

or: collapse)

For

ing

Those prog

ve this..

Send, i

hannel

Full flow control (bridge only ever full)

Full application handling (but don't *forget* x-channel)

..merging asynchronous and synchronous traditions

`XCHAN` is a new tool (in the *not* empty toolbox!)

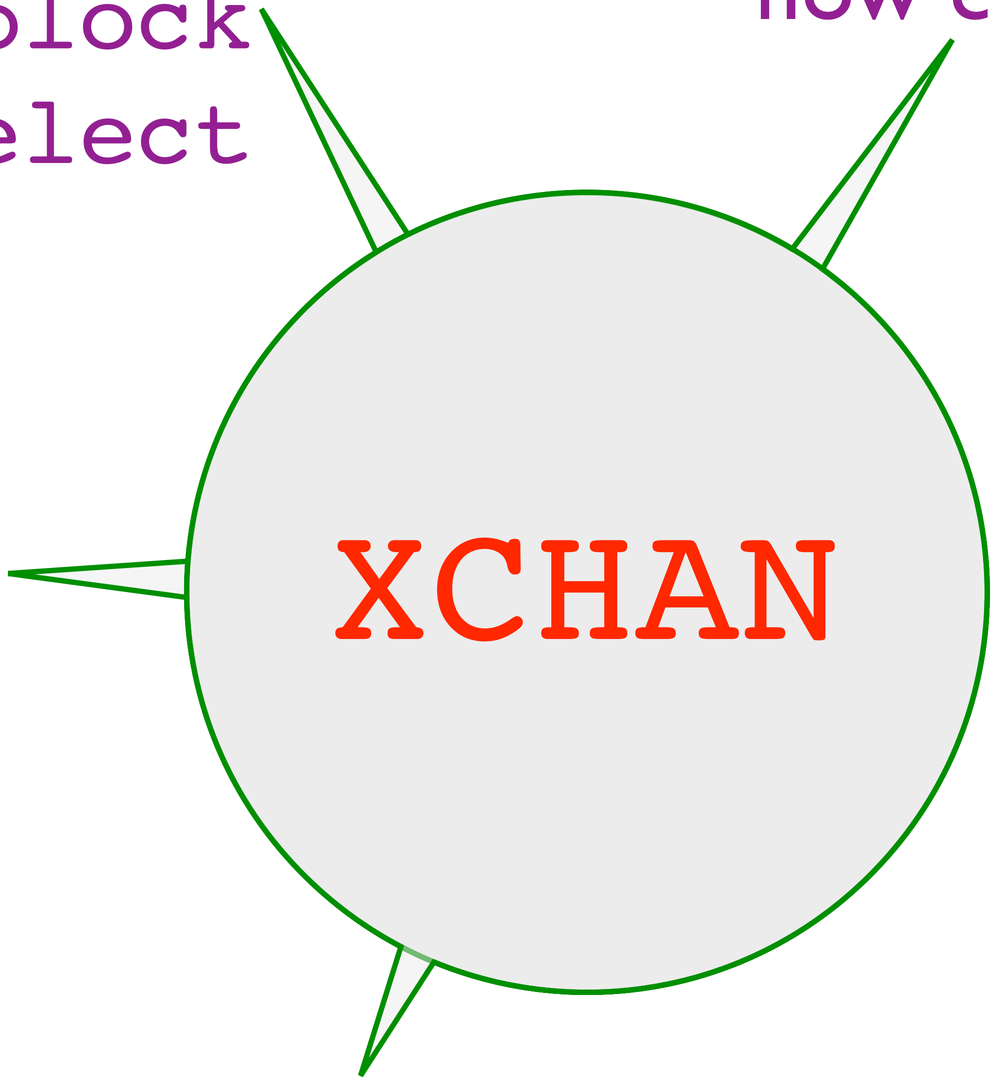
Buffering (or not)

1. buffering on-the-way:
   a. after send-and-forget (asynchronous only, no flow control)
   b. inside a buffered channel (asynchronous until full, then blocking)
   c. inside a buffered `XCHAN` (asynchronous until full, then wait for ready)

2. buffering inside a process (task, thread, …) combined with:
   a. no buffering on-the-way with zero-buffered channel (blocking synchronous, communication by synchronisation)
   b. buffering on-the-way, see bullets 1a or 1b above
   c. no buffering on-the-way with zero-buffered `XCHAN` (ready synchronous or wait for ready)

3. no explicit buffering at all (with zero-buffered channels)

Something old, something new, something borrowed, something blue - and a sixpence in her shoe..



unix
pipe
ewouldblock
select

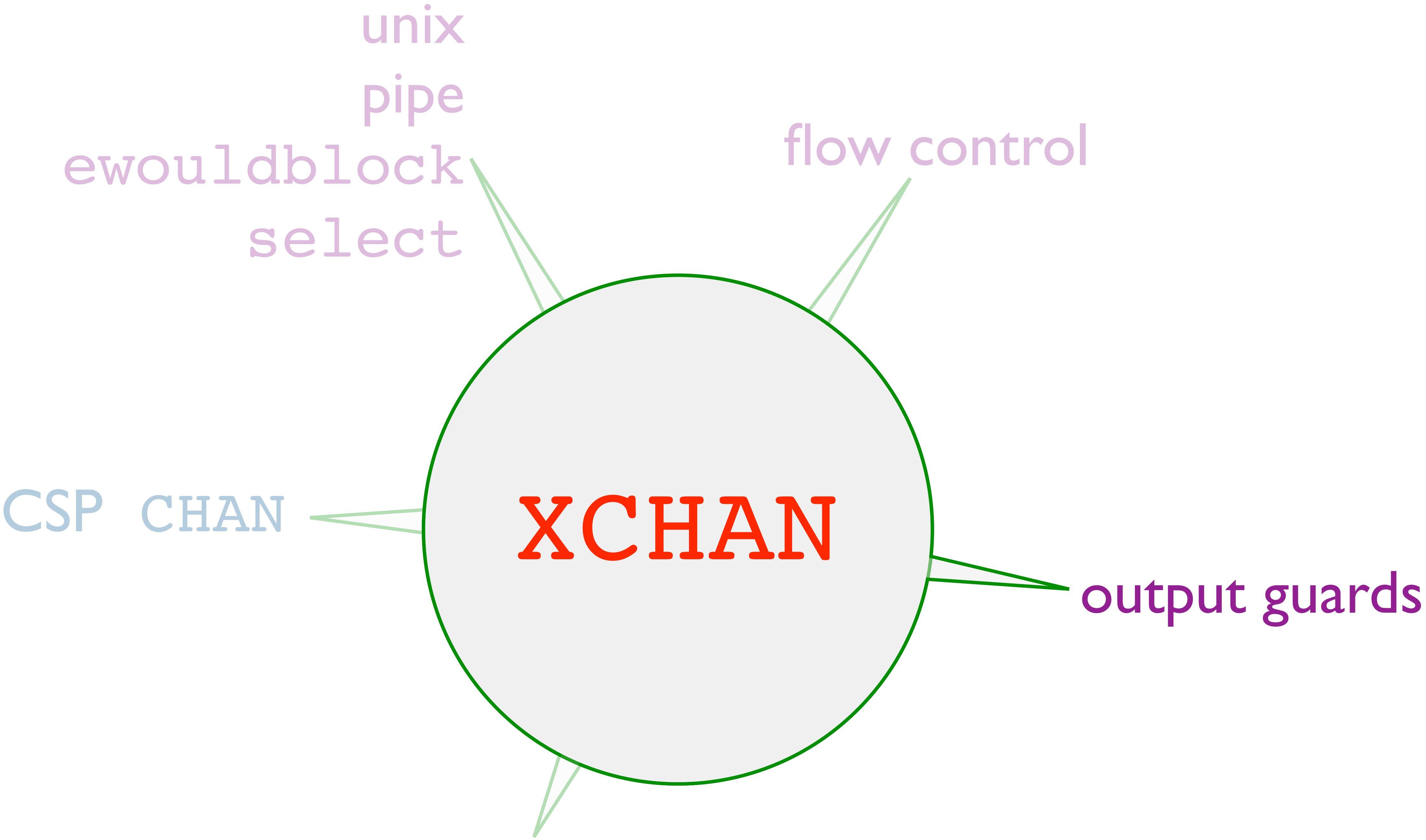
flow control

CSP CHAN

XCHAN

"If further events are to be possible (such as a channel which can report on whether or not the channel is empty) …" Schneider [10]

Something old, something new, something borrowed, something blue - and a sixpence in her shoe..



"If further events are to be possible (such as a channel which can report on whether or not the channel is empty) …" Schneider [10]

Output guard and/versus XCHAN

XCHAN	Output guard
Never blocks	May block
Would have blocked is explicit	One taken, which others could have?
Next is *sure*	Next is *attempt* and part of ALT
Commit to send, not what to send	ALT commits to what to send
Commitment is state	No such state

Output guard and/versus XCHAN

XCHAN	Output guard
Never blocks	May block
Would have blocked is explicit	One taken, which others could have?
Next is *sure*	Next is *attempt* and part of ALT
Commit to send, not what to send	ALT commits to what to send
Commitment contains *state*	No such state
A priori = "first order"	A posteriori = "second order"

Do observe

the source of the x-channel is the run-time system, as for a "timeout-channel"

Architectural leak

is when application code is added
(made more complicated)
to compensate for missing features at link level

Extra processes
Extra channels
Busy polling
Shared state
...

Fast producer, slow consumer and `XCHAN`

When Server S cannot get rid of *this* data,
it can still input more,
and finally send *newer* data

"Traditional" solution

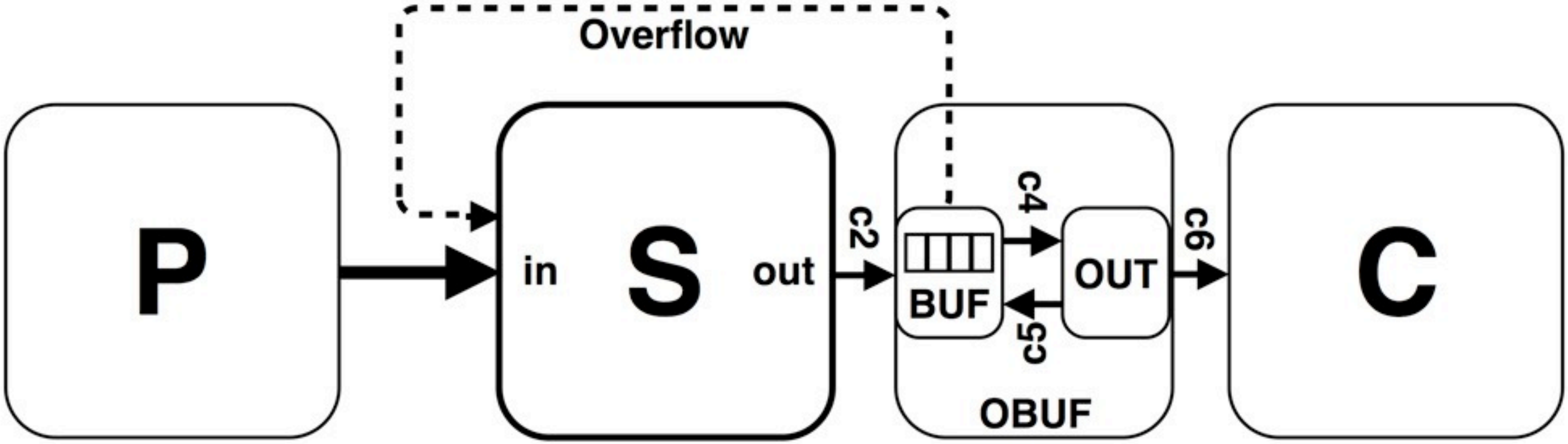


Figure 1. Example of an overflow buffer (OBUF)

An XCHAN solution

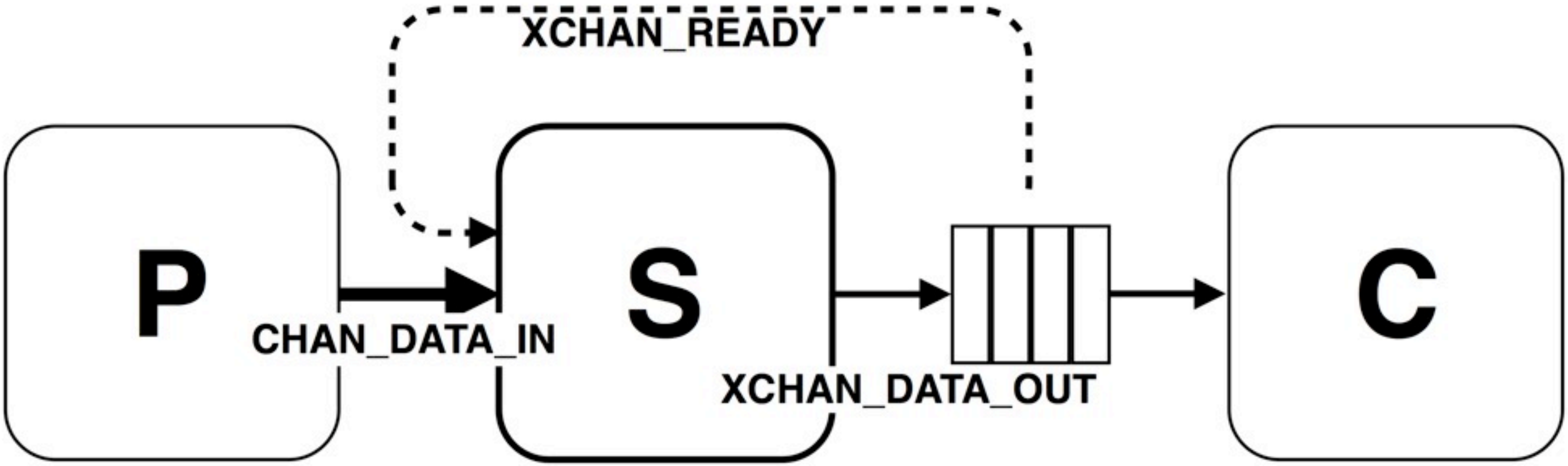


Figure 2. Buffered **XCHAN**, as shown in Listing 1 (below)

An XCHAN solution (code)

```
01 while (TRUE) {
02   ALT();
03     ALT_SIGNAL_CHAN_IN (XCHAN_READY);      // data-less
04     ALT_CHAN_IN (CHAN_DATA_IN, Value);
05?  ALT_END(); // Delivers ThisChannelId:
06
07   switch (ThisChannelId) {
08     case XCHAN_READY: {                   // sending will succeed
09!      CP->Sent_Out = CHAN_OUT (XCHAN_DATA_OUT,Value);
10     } break;
11     case CHAN_DATA_IN: {
12       if (!CP->Sent_Out) {
13         ...  handle overflow (decide what value(s) to discard)
14       }
15       else {                               // sending may succeed:
16!        CP->Sent_Out = CHAN_OUT (XCHAN_DATA_OUT,Value);
17       }
18     } break;
19     _DEFAULT_EXIT_VAL (ThisChannelId)
20   }
21 }
```

Listing 1. Overflow handling and output to buffered channels (ANSI C and macros)

Another XCHAN solution

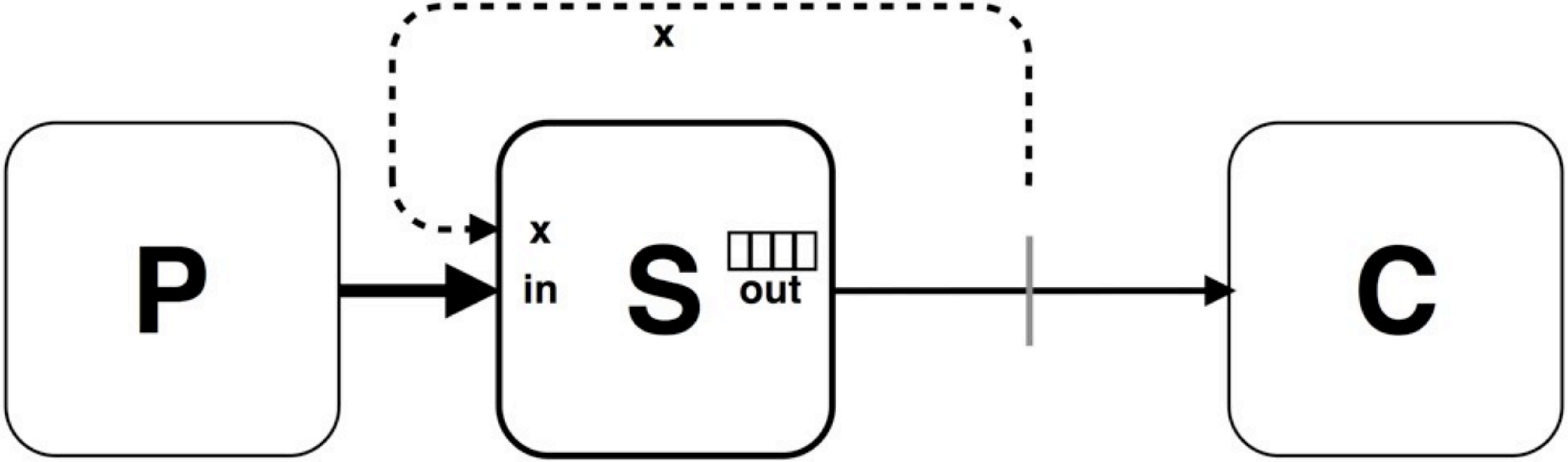


Figure 3. Zero buffered **XCHAN**

XCHANs as tool to break deadlock cycles

"Knock-come"

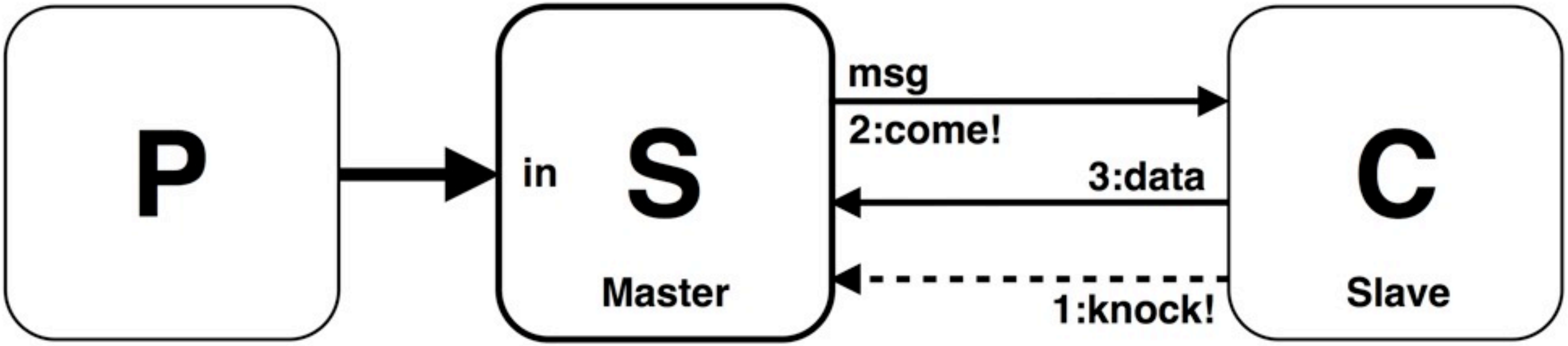


Figure 4. Traditional "knock-come" pattern

Knock-come(?) with XCHAN

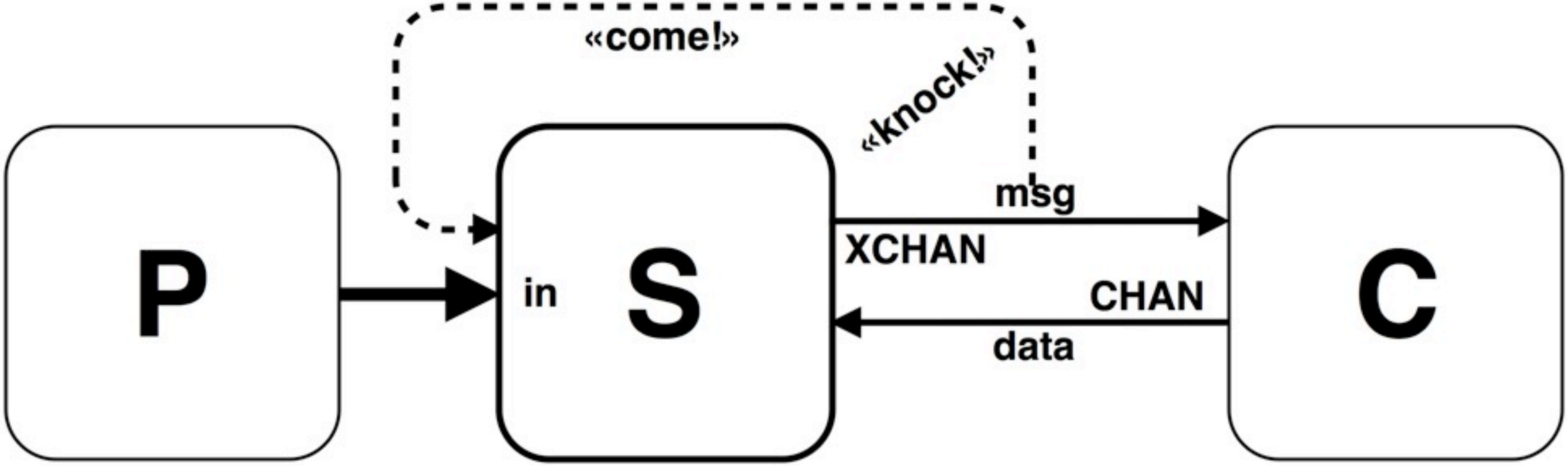


Figure 5. Same pattern with **XCHAN**

Knock-come(?) with XCHAN

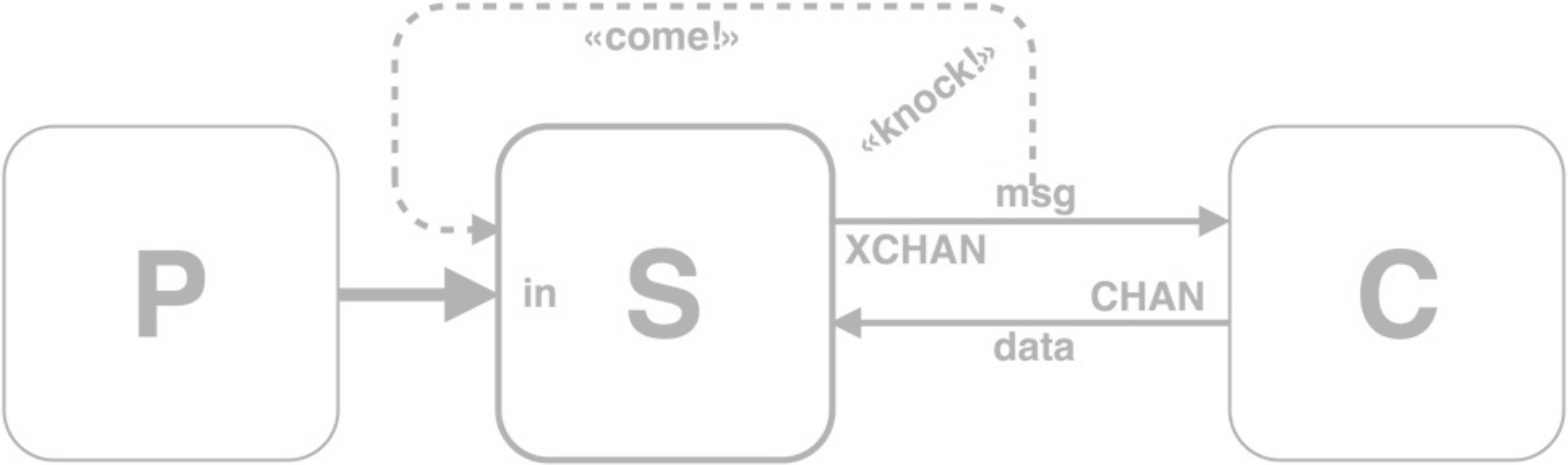


Figure 5. Same pattern with **XCHAN**

No need to think about knock-come, it comes for free!

Extending XCHANs

- `XCHAN` sending could return more than "sent" / "not sent"
- `x`-channel could deliver more than "ready"

Extending XCHANs

- `XCHAN` sending could return more than "sent" / "not sent" (like "percentage full")
- `x`-channel could deliver more than "ready" (like "closed")

Extending XCHANs

- `XCHAN` sending could return more than "sent" / "not sent" (like "percentage full")
- `x`-channel could deliver more than "ready" (like "closed")

From runtime system,
(not process)?

Semantics and implementation

- `XCHAN` that sends immediately has standard channel semantics
- `x`-channel has standard channel semantics
- Triggering of `x`-channel and intermediate blocking in receiver before sender do send, probably cannot be modeled in CSP,
  and needs help from runtime system. That was at paper time. We now know better: stay tuned

Appendix

Code courtesy of golang-

Appendix

Go has output guards

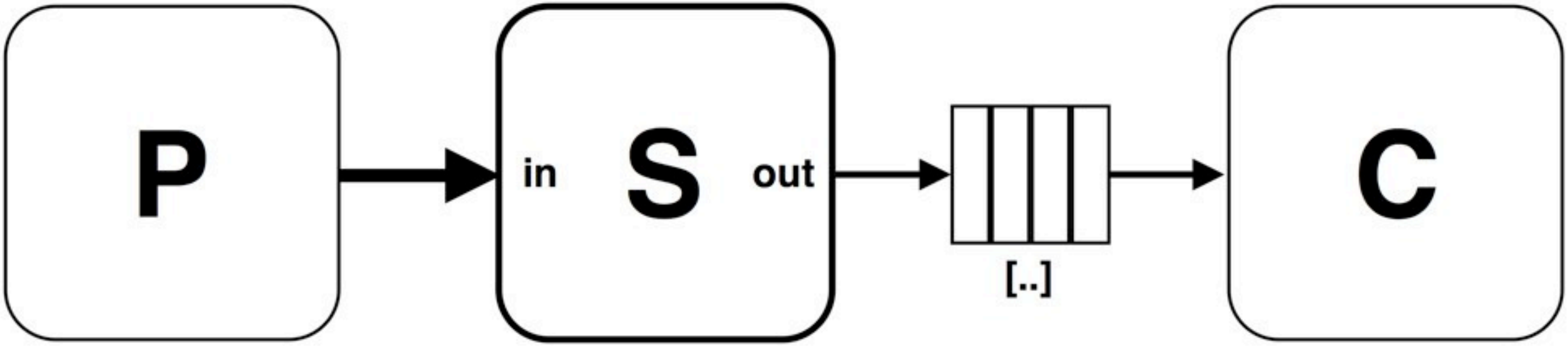


Figure 6. Go example (right channel capacity irrelevant

Appendix

```
01 func Server (in <-chan int, out chan<- int) {
02     value := 0     // Declaration and assignment
03     valid := false // --〃--
04     for {
05         outc := out // Always use a copy of "out"
06         // If we have no value, then don't attempt
07         // to send it on the out channel:
08         if !valid {
09                 outc = nil // Makes input alone in select
10         }
11         select {
12?            case value = <-in: // RECEIVE?
13                 // "Overflow" if valid is already true.
14                 valid = true
15!            case outc <- value: // SEND?
16                 valid = false
17         }
18     }
19 }
```

Listing 2. Managing without xchan in Go with goroutines

Appendix

Another code example also shown in paper

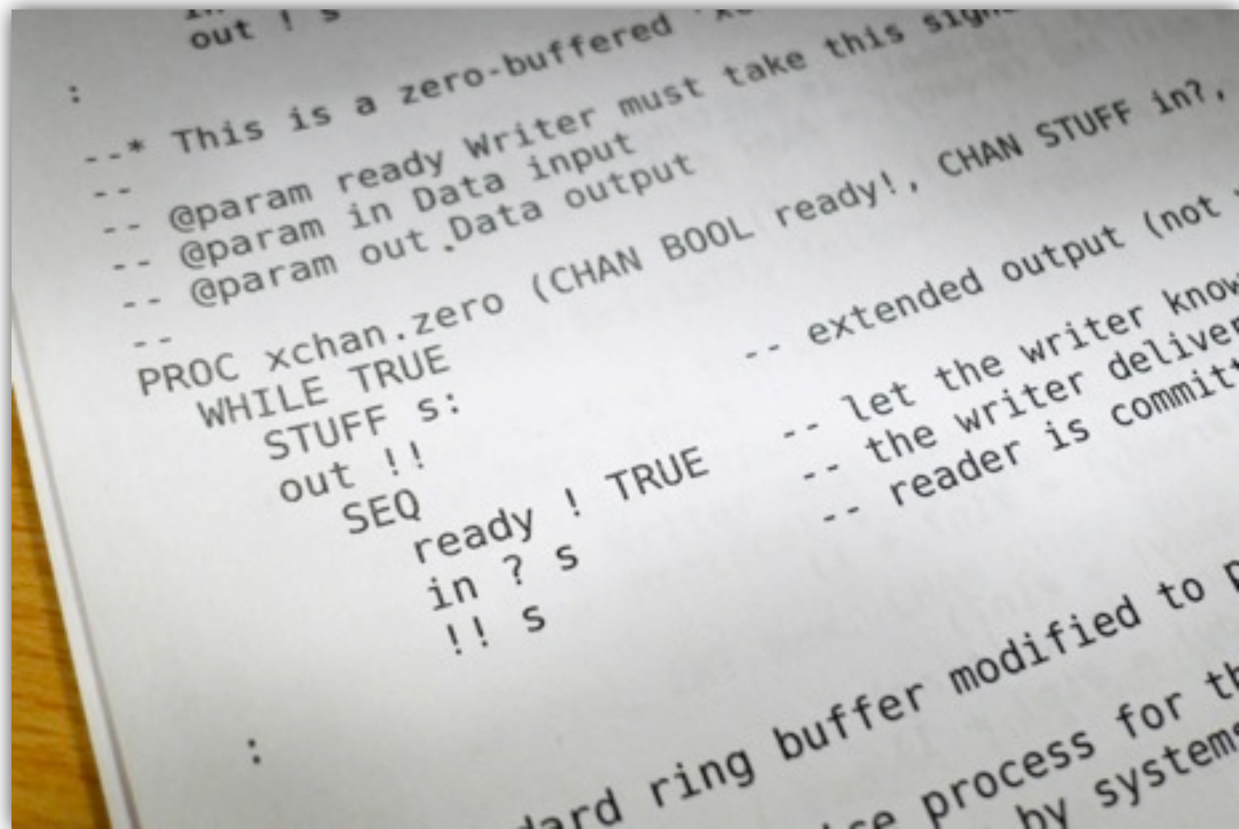
There, sender empties receiver end!
- if channel is seen to be full,

Send to itself?

We have not studied whether buffered `XCHAN` could be wrapped into the sending process, enabling the process to send to itself – but we think this is possible

Finall

Modeling XCHAN

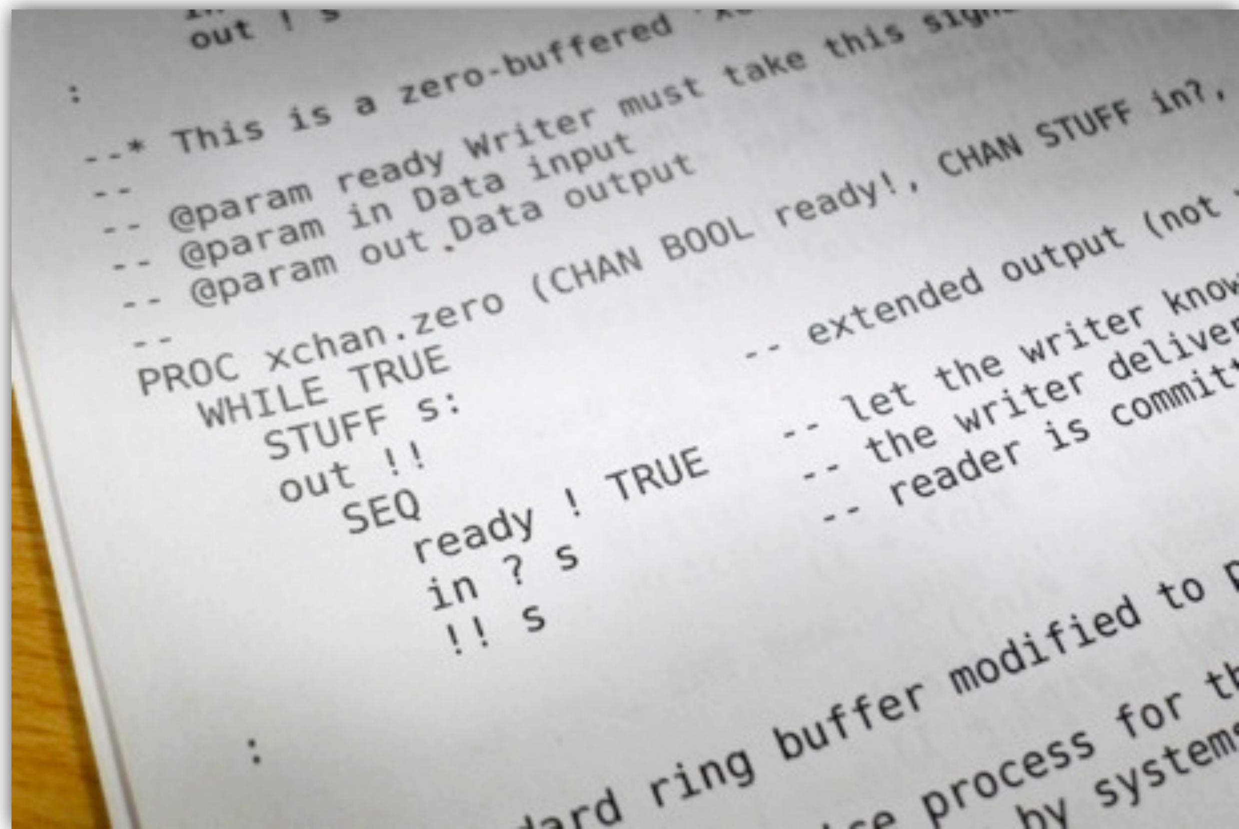


```
    out ! s
  :
  --* This is a zero-buffered
  --
  -- @param ready Writer must take this sig
  -- @param in Data input
  -- @param out Data output
  --
  PROC xchan.zero (CHAN BOOL ready!, CHAN STUFF in?,
    WHILE TRUE                    -- extended output (not
      STUFF s:
      out !!                      -- let the writer know
        SEQ                       -- the writer delive
          ready ! TRUE            -- reader is commit
          in ? s
        !! s
  :
        ard ring buffer modified to
            ce process for th
                by system
```

Modeling XCHAN

- Model of buffered XCHAN in occam-pi
- Model of **un**buffered XCHAN in occam-pi
- Done as a feasability study by the editor of the paper, Peter H. Welch, during and after the editing

Courtesy



```
    out ! s
:

--* This is a zero-buffered
--
-- @param ready Writer must take this sig
-- @param in Data input
-- @param out Data output
--
PROC xchan.zero (CHAN BOOL ready!, CHAN STUFF in?,
  WHILE TRUE
    STUFF s:
    out !!                          -- extended output (not
      SEQ
        ready ! TRUE                -- let the writer know
        in ? s                      -- the writer deliver
      !! s                          -- reader is commit
:

    dard ring buffer modified to p
    ce process for th
    by system
```

Thank you!

¿questions?

First page: Drammen signal box (1910), now at the Railway Museum at Hamar

**Three pictures: Vemork hydroelectric plant Rjukan (1911),
now the Norwegian Industrial Workers Museum (2012)**

Small toolbox that I made for Isac