The External Recovery

Arkadiusz D. Danilecki

Anna Kobusińska

Mateusz Hołenko

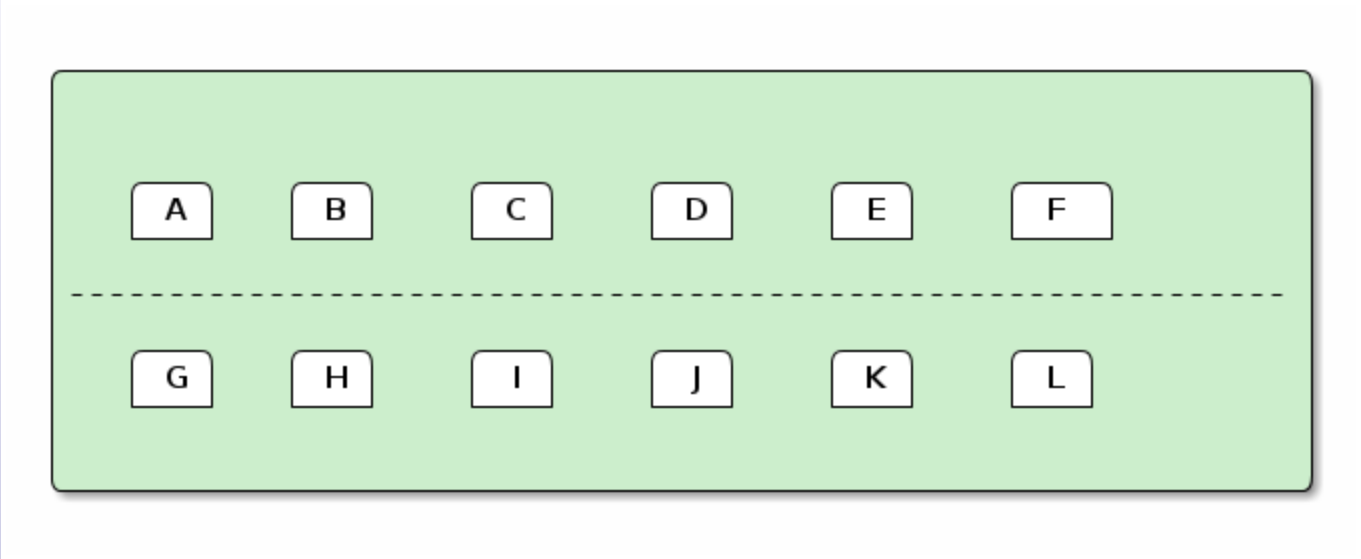
Piotr Zierhoffer

Poznań University of Technology

This work was supported by the Polish National Science Center under Grant No. DEC-2011/03/D/ST6/01331

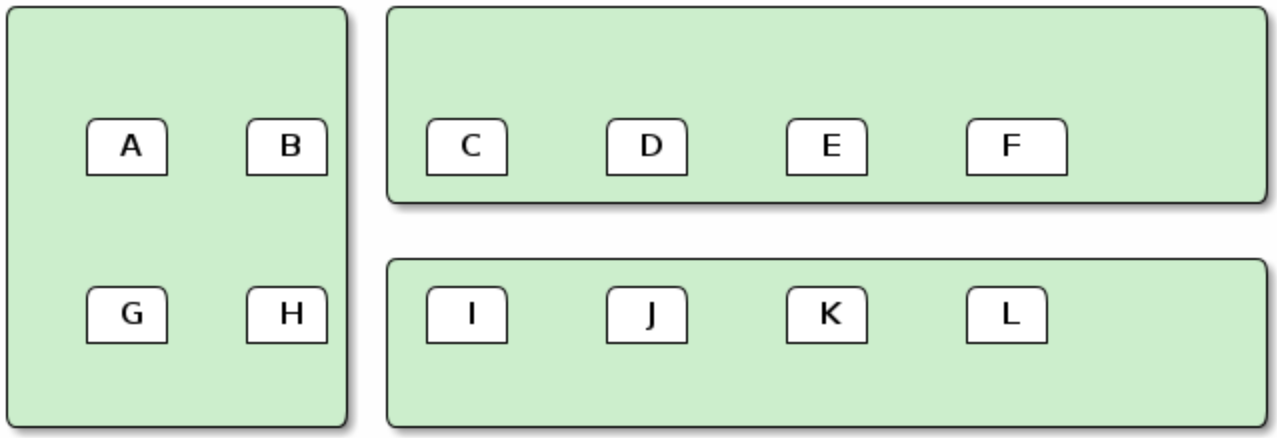
The motivation

- Large systems are hard to maintain



The motivation

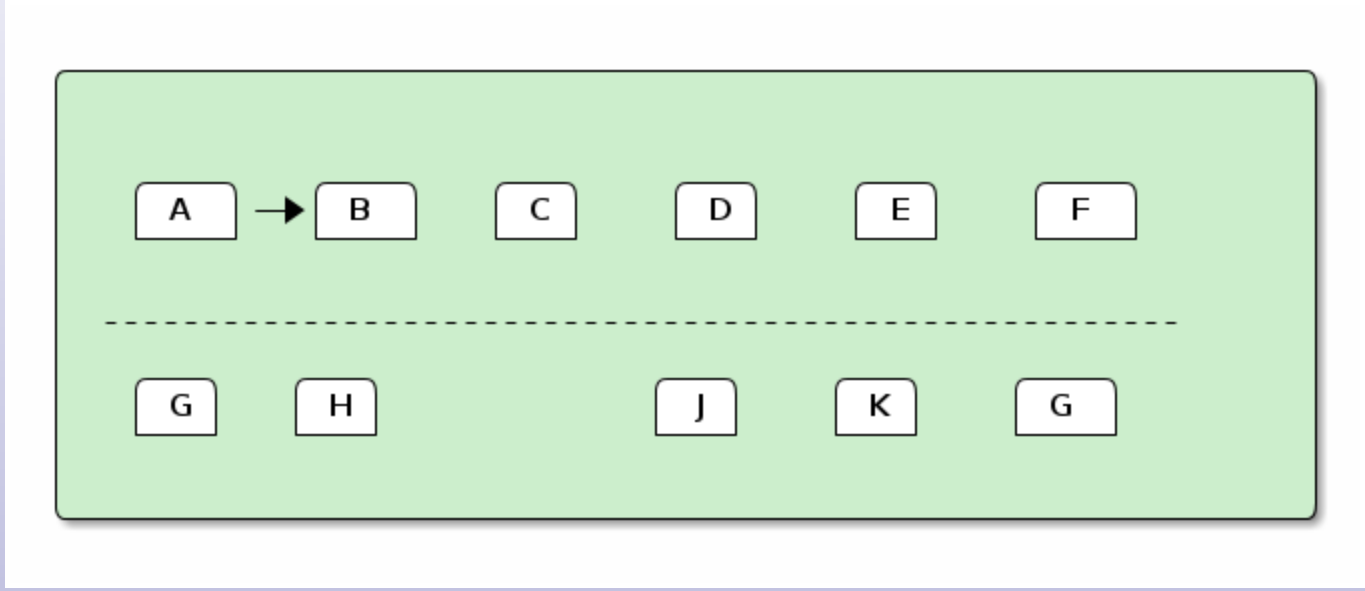
- Large systems are hard to maintain



Solution
Divide a system into a lot of small, independent components, and instead of forcing policies, establish conventions

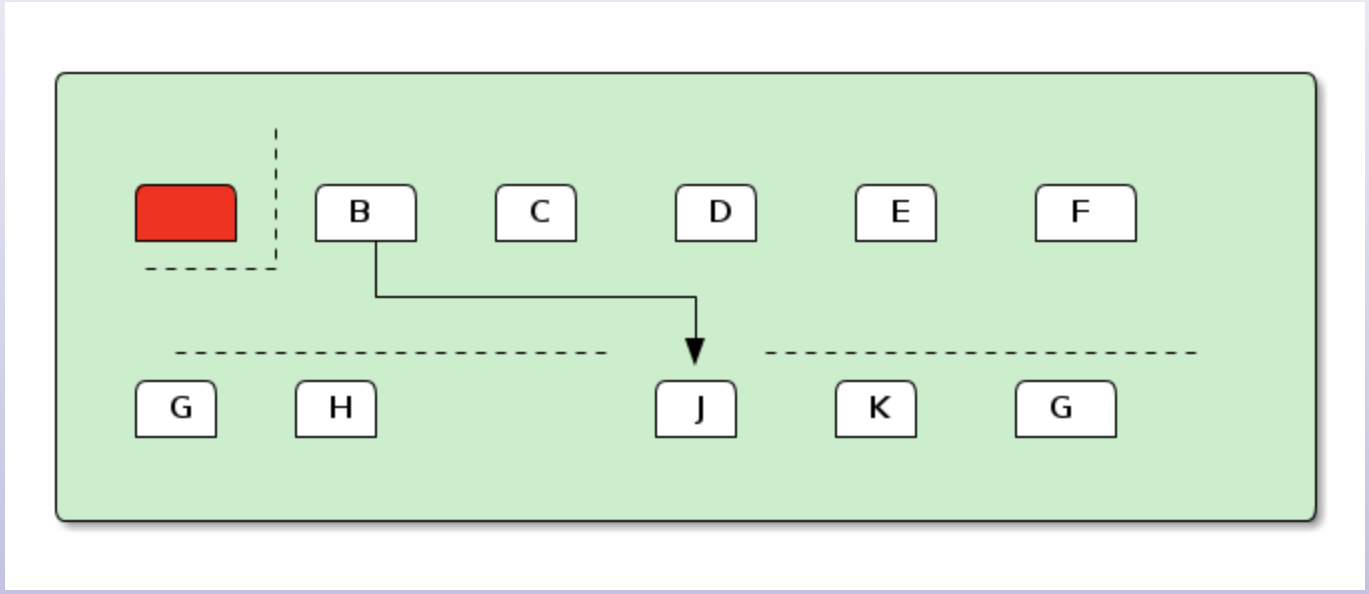
Motivation continued

- Large systems are error-prone
- During execution, dependencies are created between nodes in different subsystems



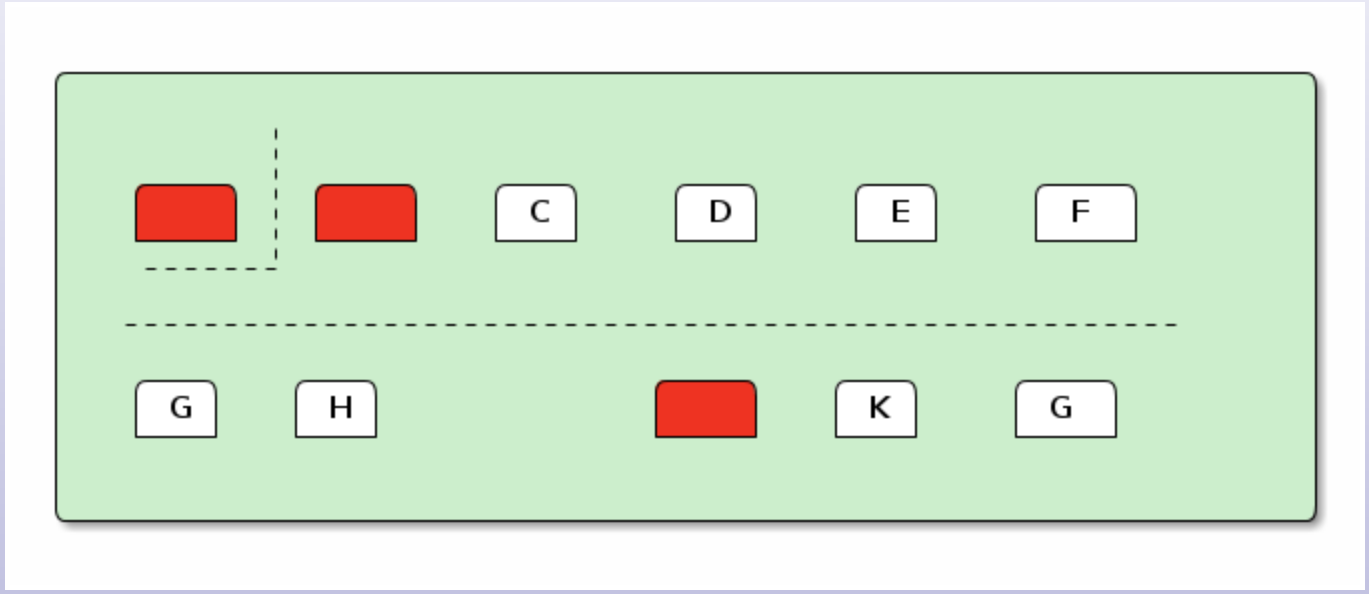
Motivation continued

- Large systems are error-prone
- During execution, dependencies are created between nodes in different subsystems



Motivation continued

- Large systems are error-prone
- During execution, dependencies are created between nodes in different subsystems



Motivation continued

- Large systems are error-prone
- During execution, dependencies are created between nodes in different subsystems
- How to prevent effects of one node's crash to spill over to other parts of system?

> ***Solution***:
> *Log messages exchanged between nodes belonging to different subsystems*
> ***External recovery***:
> *System is recovered from the **outside**, using only logged messages which were sent/received to/from other subsystems.*

A problem

What are the minimal restrictions which must be put on subsystem's independence in order to make the external recovery possible?

> *Question provoked by our earlier research on recovery in SOA systems, starting in 2009:*

- *Maybe we should care not just about algorithm's performance, but also on how much restriction an algorithm puts on independence of recovered system?*

The nature of the restrictions

- ***Behavior restriction***

  Before-hand imposed restrictions, allowing only for some possible history executions; e.g. what amount of determinism is allowed; who can send what to whom; etc.

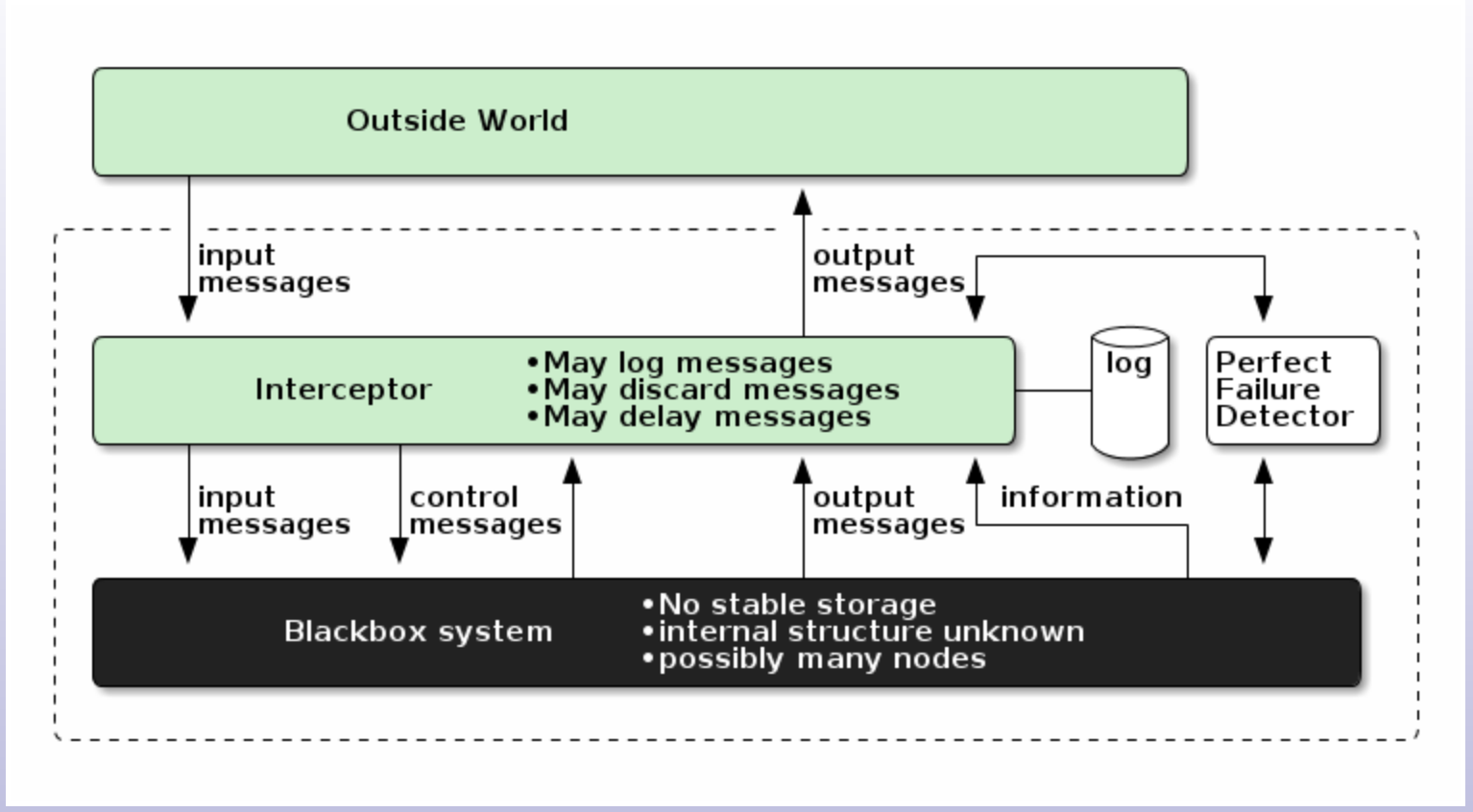
- ***Requiring cooperation***

  Subsystem must be aware of the external recovery; control messages are exchanged or messages contain data relevant only for the recovery; execution during the recovery may be different from normal execution

- ***Requiring information***

  Subsystem structure must be known, e.g. number of nodes or their roles; history of execution is made known to the outside world (e.g. who sent what to whom)

System Model



Causal and Truly Causal relations

$$send_i(m_1) \mapsto receive_j(m_2)$$

$$event_i^k \mapsto event_i^{k+1}$$

$$event_i^p \mapsto event_j^l \wedge event_j^l \mapsto event_k^m \Rightarrow event_i^p \mapsto event_k^m$$

Note

Similarly we define causal relation on process' states

Some causal relations are not dictated by application logic. But there are also events *truly caused* by some others (Tarafdar, Garg 1998), where relation of *true causality* is de facto a reflection on application logic.

Inputs and sessions

- ***Input:***

  Message sent to subsystem from the outside world

- ***Session of input message *m*:***

  All events truly caused by m

We assume all events belong to some session and that sessions do not overlap

Outputs and reactions

- ***Output:***

  Message sent from a subsystem to the outside world

- ***Reaction to input message *m* :***

  All outputs, whose send events were truly caused by m

We assume all events belong to some session and that sessions do not overlap

Consistent state and externally visible state

- ***Consistent State**:*

  Informally, a global state which could possibly happen in some execution without a crash at some global time t.

- ***Externally equivalent states**:*

  For any given output, it can be a product of only limited set of global states. Those states are externally not-distinguishable - they are equivalent from the points of the outside world (observable states, Bostan, Atkinson, 2009)

Failure model

- Fail-stop
- After crash all are restarted from initial, consistent state
- Failure detector notifies interceptor about restart
- Interceptor does not crash (or it crashes, but is transparently recovered)
- Crashes are rare
- Orphaned messages are detected and discarded (e.g. using epoch numbers)

Determinism

- ***Piecewise determinism***

  Non-deterministic events (usually only receive) have their determinants. You can capture all necessary determinants to replay the execution. Order of receive events is important.

- ***Send-determinism***

  Informally, ordering of receive events is not important (but what is received IS important), send events are always the same given the same messages are received.

- ***Channel-determinism***

  Informally, ordering of receive is not important, send events along some channel are always the same (but ordering along different channels may be different).

Restricting the behavior

Restricting the behavior (1)

Imagine we want to be able to treat a subsystem from the outside as a single, PWD node.

Note

If processes in $\mathcal{P}$ works under PWD, then they cannot be externally recovered unless they either cooperate with I or expose information about its behavior.

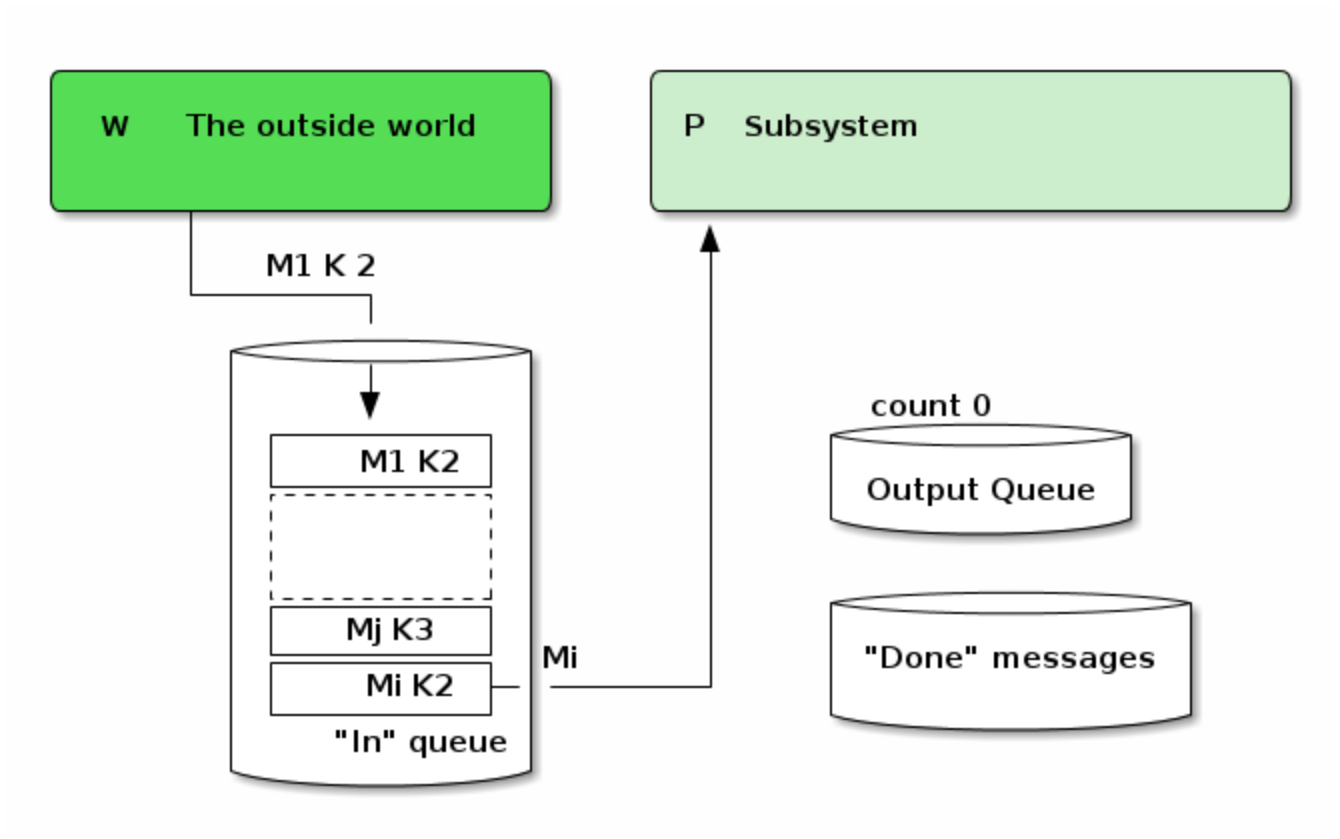
Restricting the behavior (2)

- Within each session, processes are send-deterministic
- Each session produces a known a priori number of k outputs (different for every input) and no event happens within a sessions when k's output is sent to the outside world.

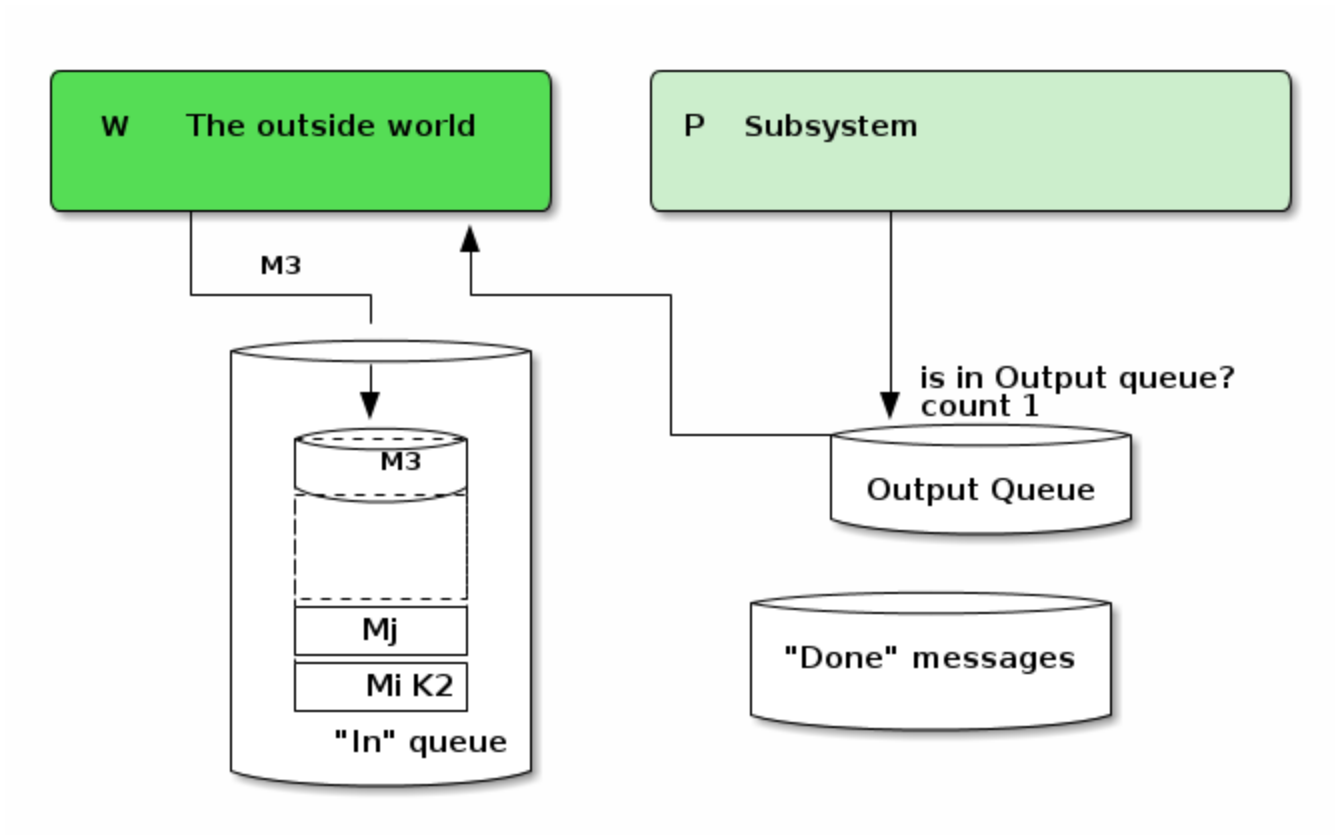
Note

One can abandon the second restriction, but that would require cooperation with $\mathcal{P}$

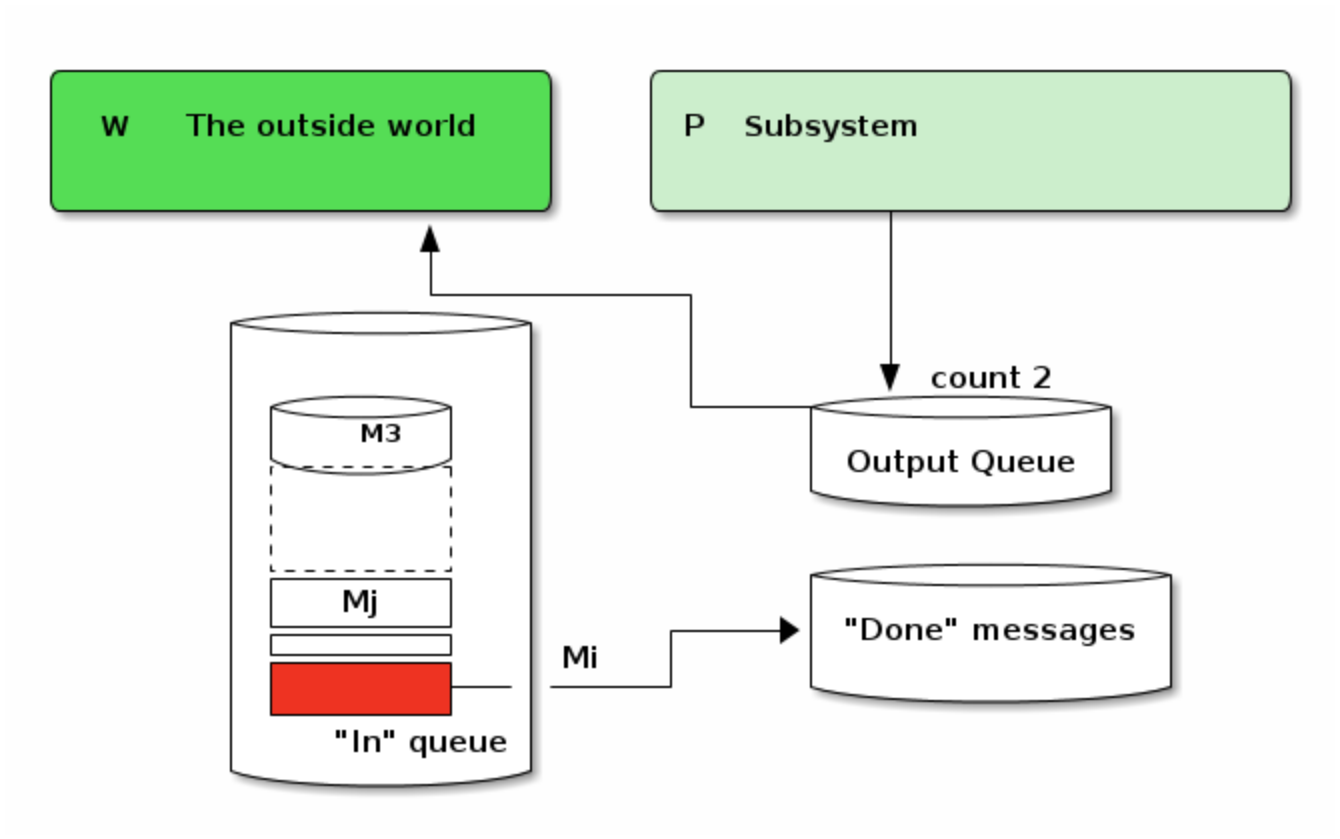
Algorithm


W The outside world
P Subsystem
M1 K 2
M1 K2
Mj K3
Mi K2
"In" queue
Mi
count 0
Output Queue
"Done" messages

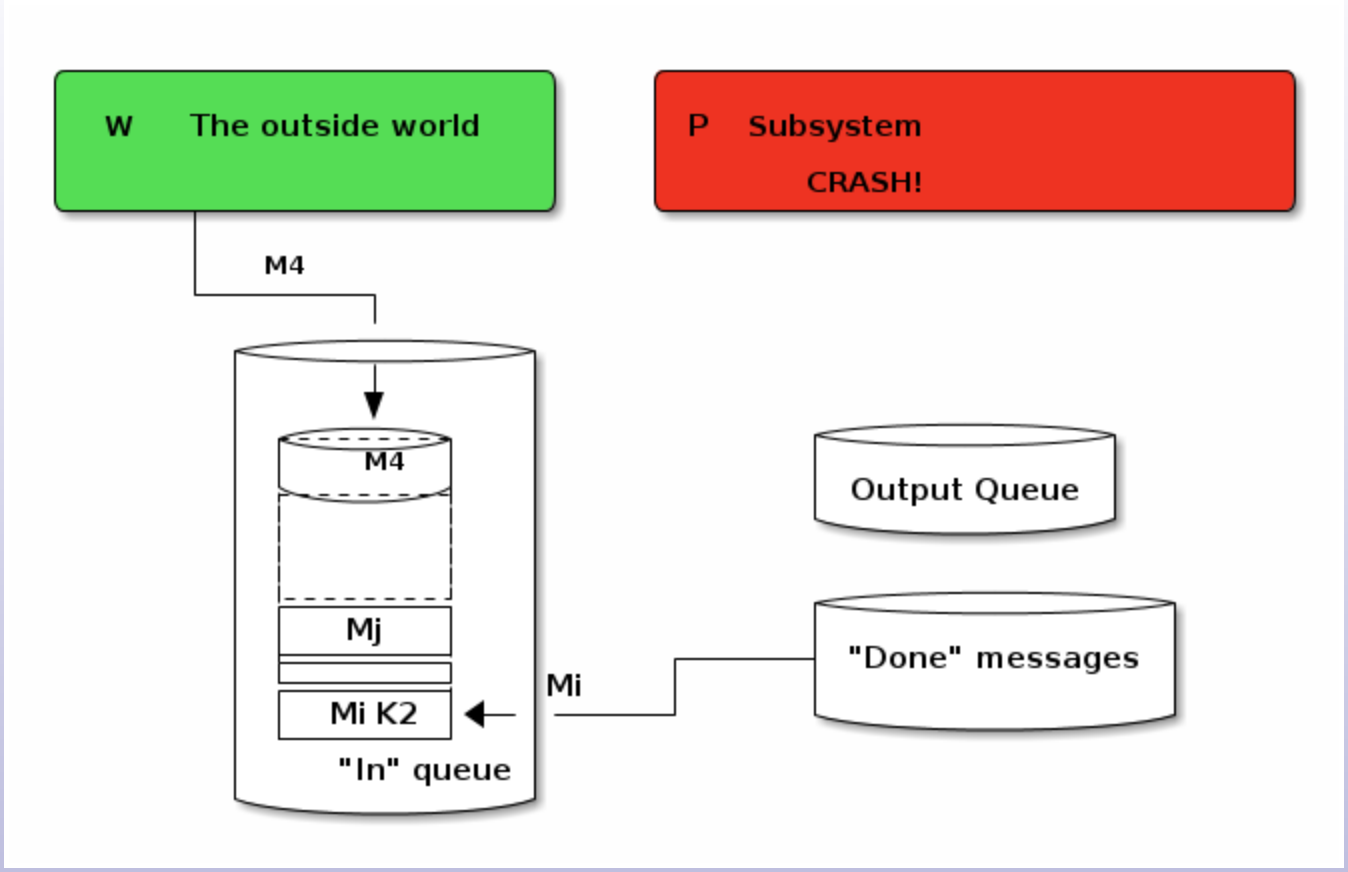
Algorithm


W The outside world
P Subsystem
M3
is in Output queue?
count 1
M3
Output Queue
Mj
Mi K2
"In" queue
"Done" messages

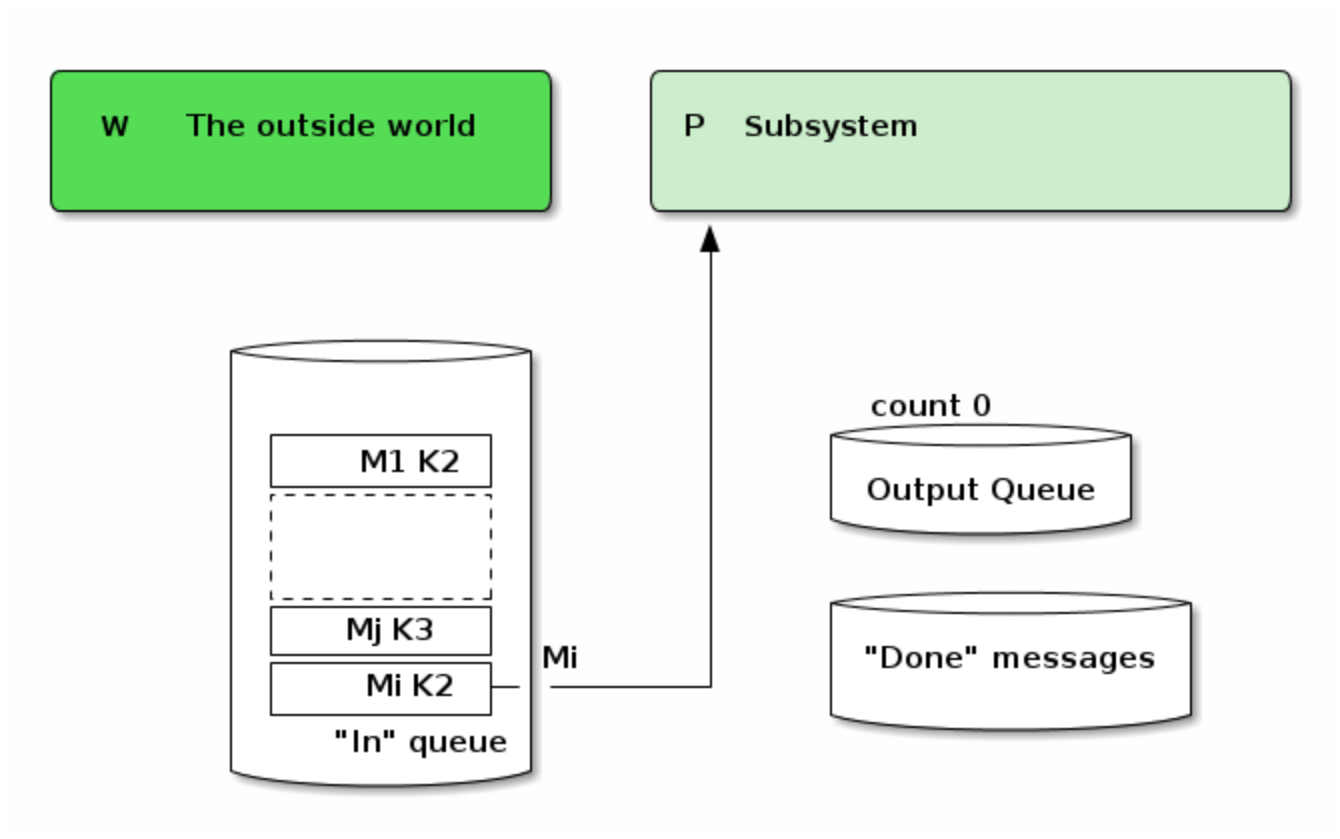
Algorithm


W The outside world
P Subsystem
count 2
Output Queue
M3
Mj
"In" queue
Mi
"Done" messages

Algorithm


W The outside world
P Subsystem
CRASH!
M4
M4
Mj
Mi K2
"In" queue
Mi
Output Queue
"Done" messages

Algorithm


W The outside world
P Subsystem
count 0
Output Queue
M1 K2
Mj K3
Mi K2
Mi
"In" queue
"Done" messages

Algorithm

```
var Q:   queue of messages
var L:   queue of messages
var Out: queue of messages
var state: enum { busy, ready} ← ready
var count: integer

when received M from W at I  // M from W is received at interc
 Q ← Q ∪ M // M is appended at the end of Q

when Q ≠ 0 ∧ state = ready at I
 state ← busy
 M ← Q.front
 count ← 0
 send M to P // P is destination of M within a subsystem
```

Algorithm cont.

```
 when detected restart( P ) at I
  Q ← L ⋃ Q // prepends L to Q
  state ← ready

when received M from P at I // M from P is received at interce
  if M ∉ Out then
     Out ← Out ⋃ M
     forward M
  else
     discard M
  end if
  count ← count + 1
  if count = K(M) then // K(M) returns known a priori number c
     M ← Q.front
     Q ← Q \ M
     state ← ready
     if M ∉ L then
        L ← L ⋃ M
     end if
   end if
```

Discussion

- No cooperation required between P and I
- No informations exposed to I
- *Minimal* restrictions on behavior of P - either restrictions have to be stronger, or informations would have to be exposed
- No difference between normal execution and recovery
- Only for applications with call-return pattern
- After crash, the internal structure of P may change: some nodes may be permanently removed, some may be added
- An externally equivalent state of P may be replicated on other subsystem with different number of nodes etc.
- As it serializes processing both during normal execution and during the recovery, obviously it could not have impressive performance

Restricting more than one axis

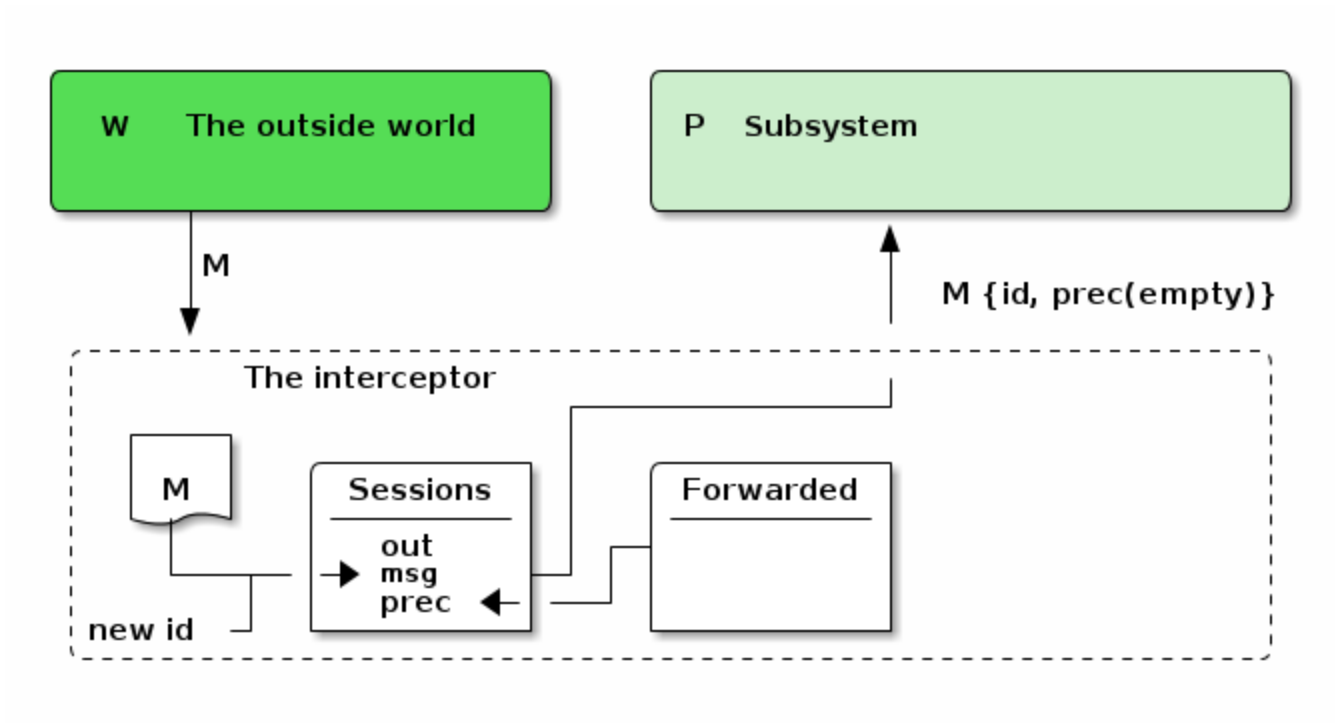
Restrictions

- Processes are send-deterministic within each session
- Sessions are serializable (many serializations could be possible) - explained on a next slide
- Size of reaction to M is known a priori (but sessions may still continue after sending last output)
- Whenever a process receives a message within a session, it must eventually send a message (possibly, an output)
- Messages have additional fields: session identifier and identifiers of all preceding sessions.
- Information about session serialization is exposed.

Serializable sessions

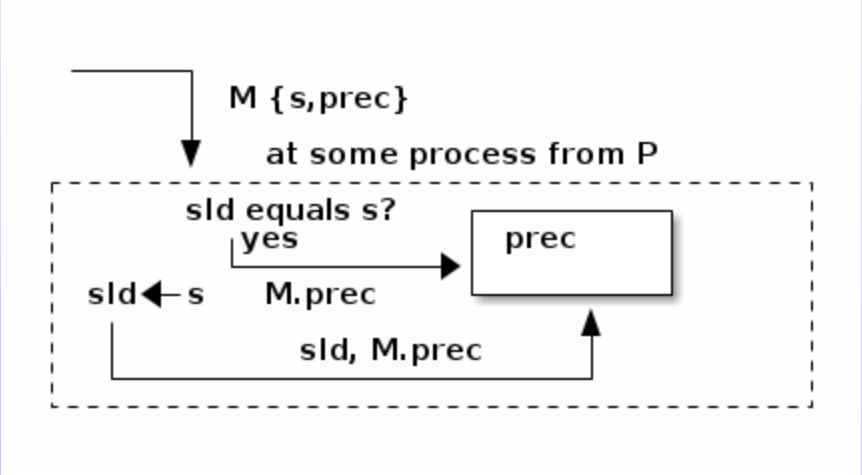
- When P participates in session A, and then participates in session B, then A < B on P
- If not A < B or B < A on some P, then A || B
- If A < B on some P, then on all P, A < B, or A || B

Algorithm

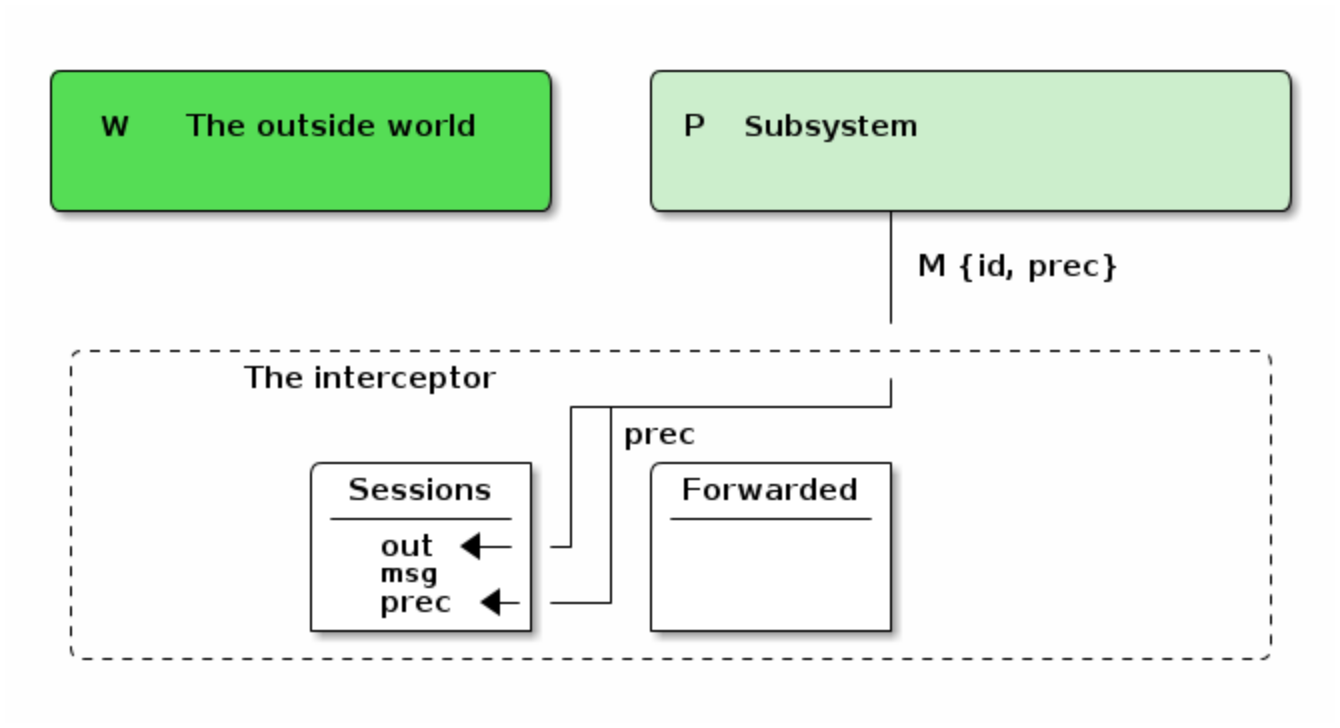

W The outside world
P Subsystem
M
M {id, prec(empty)}
The interceptor
M
Sessions
out
msg
prec
Forwarded
new id

Algorithm

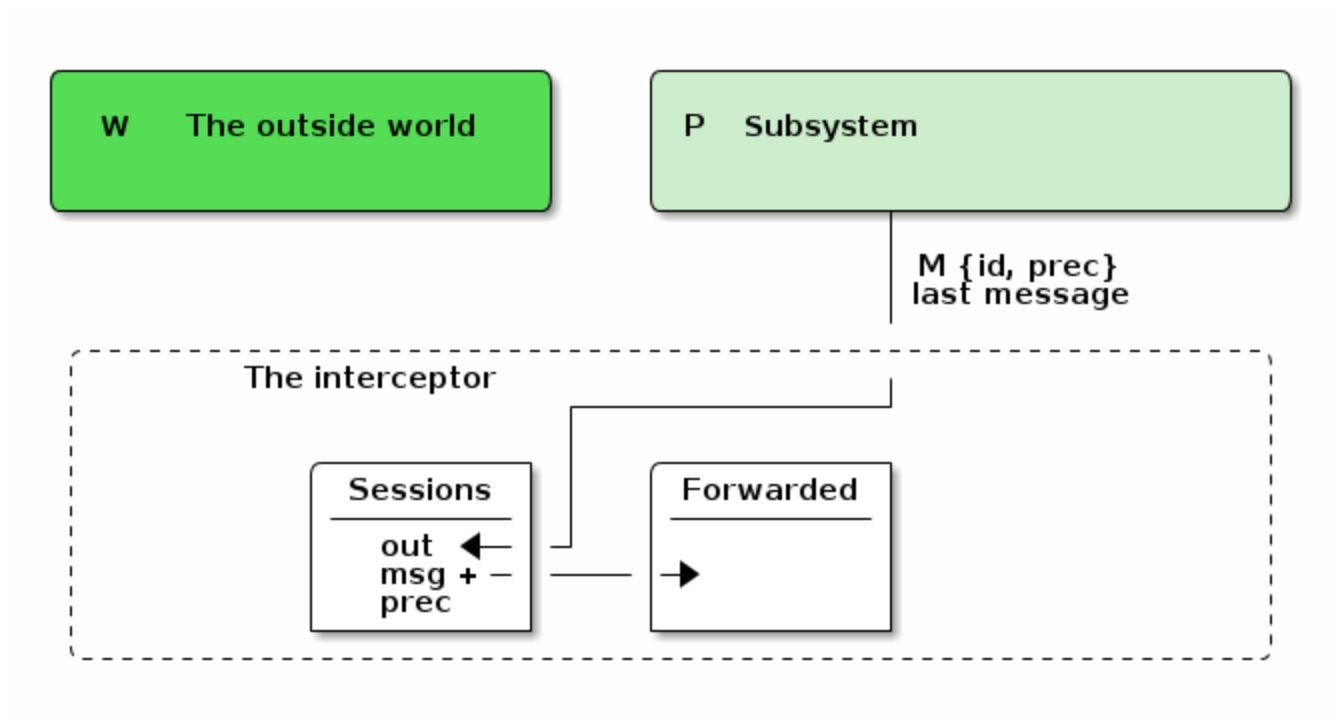
- Each process must maintain prec variable (initially empty) containing session identifiers
- Each message has two fields: id of the session and prec field
- When message is sent, variable prec is sent along as a prec field



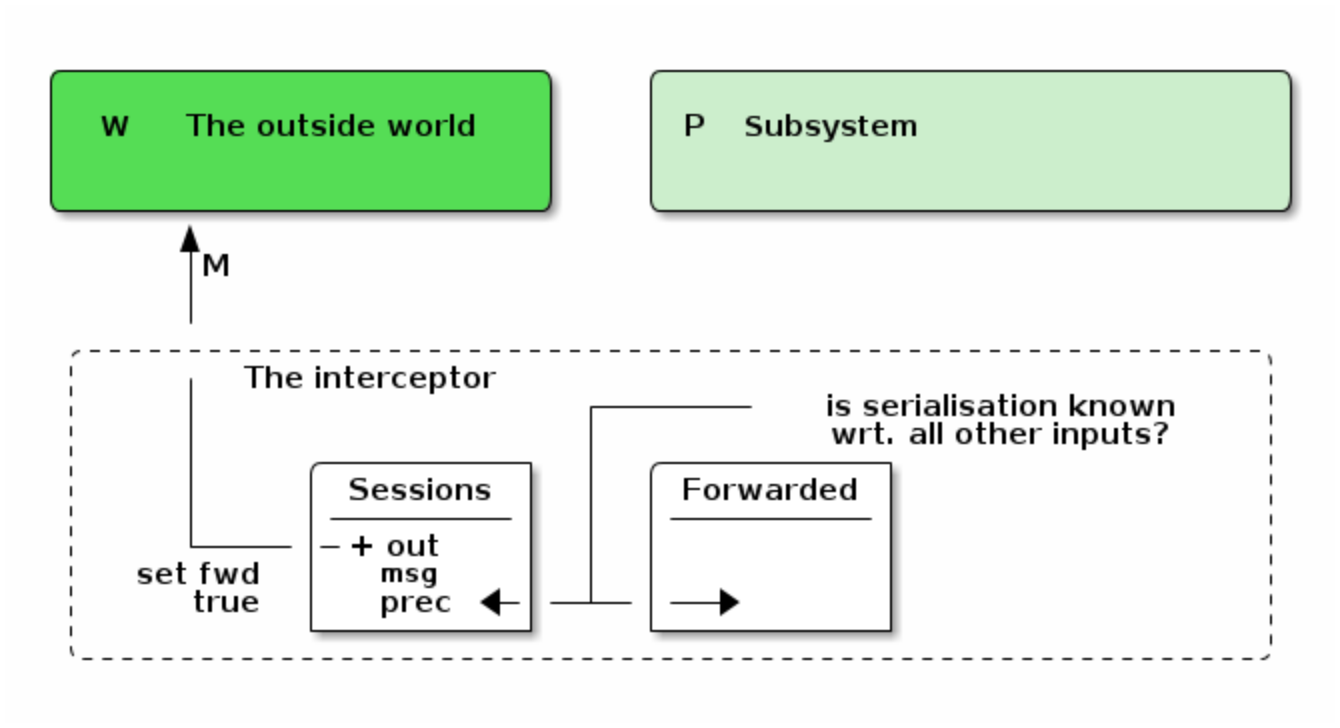
Algorithm


W The outside world
P Subsystem
M {id, prec}
The interceptor
prec
Sessions
out
msg
prec
Forwarded

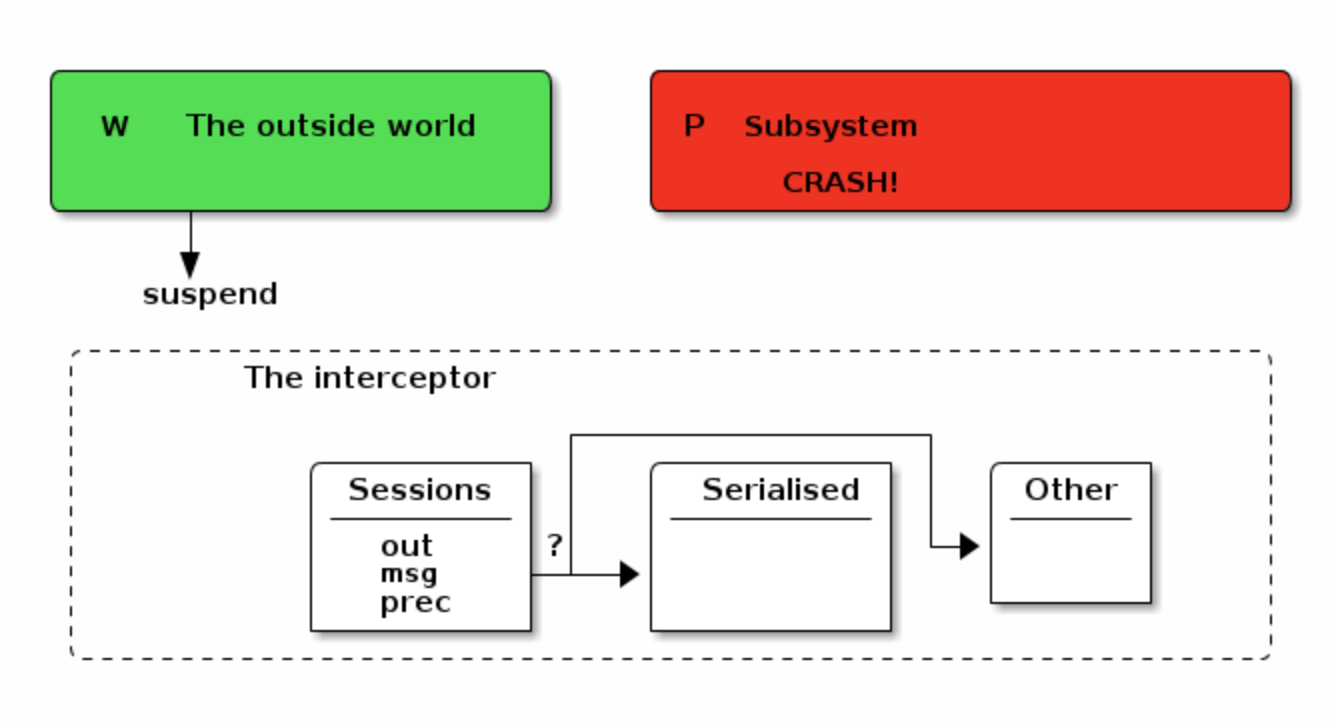
Algorithm


W The outside world
P Subsystem
M {id, prec}
last message
The interceptor
Sessions
out
msg + –
prec
Forwarded

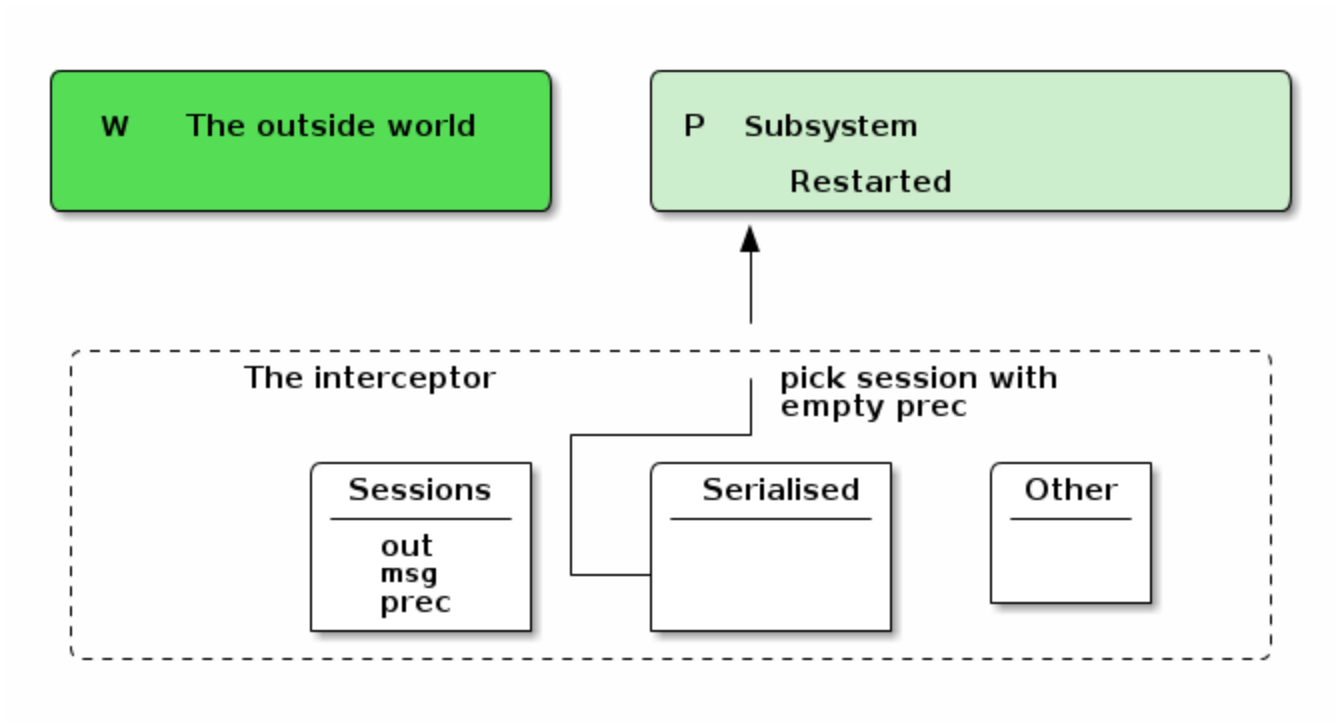
Algorithm


W The outside world
P Subsystem
M
The interceptor
is serialisation known
wrt. all other inputs?
Sessions
– + out
msg
prec
Forwarded
set fwd
true

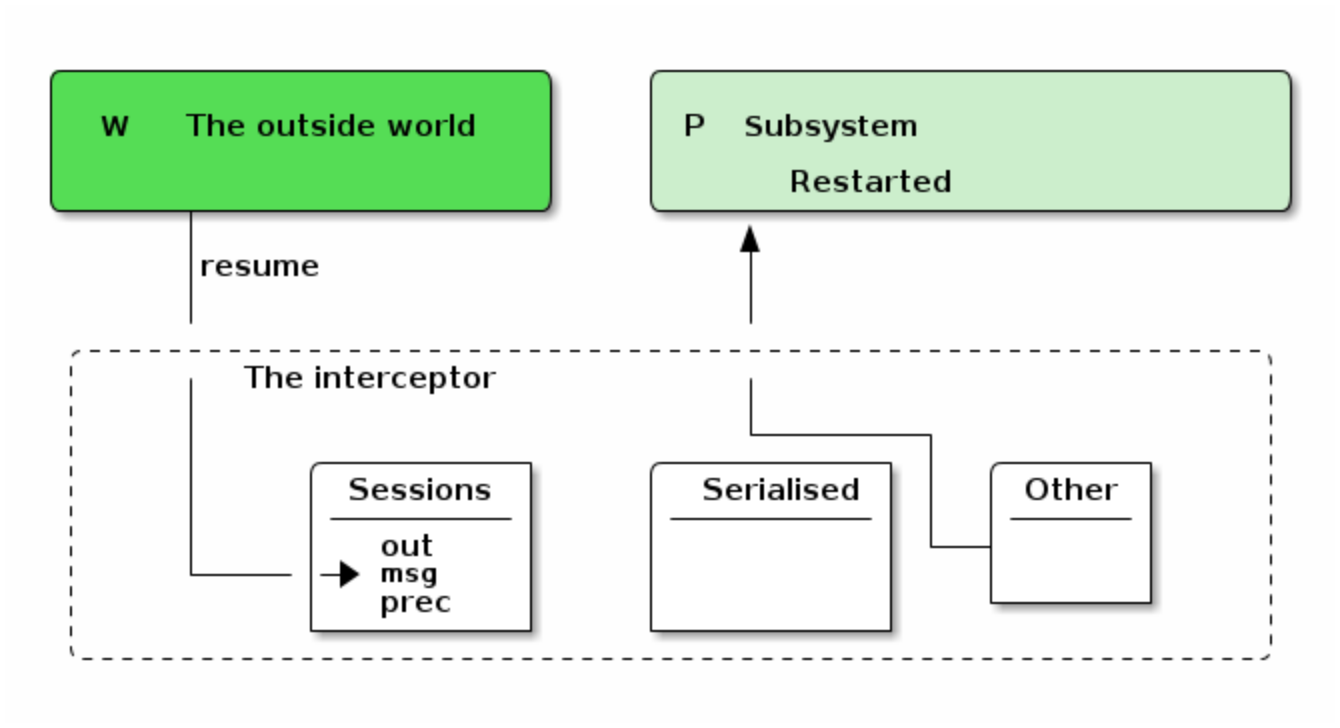
Algorithm


W The outside world
P Subsystem
CRASH!
suspend
The interceptor
Sessions
out
msg
prec
?
Serialised
Other

Algorithm


W The outside world
P Subsystem
Restarted
The interceptor
pick session with
empty prec
Sessions
out
msg
prec
Serialised
Other

Algorithm


W The outside world
P Subsystem
Restarted
resume
The interceptor
Sessions
out
msg
prec
Serialised
Other

Algorithm

```
message PACKET is record of
    id: integer // session identifier
    prec: set of integers
    msg: data // original message

Session is record of
    msg: message // input which started the session
    sId: integer // session identifier
    prec: set of integers // preceding sessions
    out: set of messages  // reaction to msg
    fwd: boolean

Variables
    var M^i: set of Sessions
    var F : set of integers // finished sessions
    var M': set of Sessions // serialized sessions (during rec
    var M^o: set of Sessions // not yet fully serialized sessic
    var x : Session
    var pkt: PACKET
    var mode: enum { ready, busy } ← ready
```

Algorithm

```
when received M from W at I
    x.msg ← M
    x.sId ← "new unique session identifier"
    x.prec ← F
    x.out ← ∅
    M^i ← M^i ∪ x
    pkt.data ← M
    pkt.sId ← x.sId
    pkt.prec ← ∅
    wait until mode = ready
    send pkt to P

when received M from P at I
    x ← { x: x∈ M^i ∧ x.sId = M.sId }
    M^i ← M^i \ x
    if M.data ∉ x.out then
        x.prec ← x.prec ∪ M.prec
        x.out ← x.out ∪ M.data
        if |x.out| = K(M) then
            F ← F ∪ x.sId
        end if
```

Algorithm

```
when ∃x∈M^i : x.sId ∈ F ∧ x.fwd = false ∧
        ∀y∈M^i, y.sId∉F⇒ x.sId∈ y.prec ∨ y.sId ∈ x.prec
    x.fwd ← true
    foreach msg ∈ x.out do
        send msg to W
    end for
```

Algorithm

```
when detected restart(P) at I
    mode ← busy
    M' ← {x: x∈ M^i ∧ x.fwd=true}
    M^o ← M^i \ M'
    while M' ≠ ∅ do
        foreach x∈M' do
            if x.prec = ∅ then
                break // at least one such element
                 // must exist, since sessions are serializable
            end if
        end for
        pkt.msg ← x.msg
        pkt.prec ← ∅
        pkt.sId ← x.sId
        send pkt to P
        for i ∈ {1..K(M)} do
            receive pkt from P
            discard pkt
        end for
        M' ← M' \ x
        foreach x∈ M^o do
```

```
end for
mode ← ready
```

Discussion

- Serializes execution during recovery; expect performance penalties. Can be avoided if P cooperates with I (message m's processing is suspended if m.prec $\not\subseteq$ prec
- Session serialization does not have to be repeatable; replicable serialisation enforced during recovery
- Does not require knowledge about structure of P (number of nodes) nor exact history (who sent what to whom)

Conclusions

Conclusions

- Algorithms graded not just by the performance, but also but the restrictions they put on a system
- More flexibility by imposing restrictions (e.g. number of nodes can be changing), this flexibility could be exploited by applications
- You can restrict only one aspect of independency (behavior)

Other results

Things which didn't fit in a paper

- Manetho based (exposing information about history and number of nodes, requires cooperation and tagging messages, PWD processes)
- Event-logger inspired (exposing information about history, number of nodes, requires cooperation. Maintaining logs with message ids, flushed to interceptor each time process sends a message, complicated recovery)
- Forced determinism (transformation of PWD into quasi-deterministic with a priori total order messaging primitives)

Future work

- Implementation
- Performance tests
- Other minimal results?
- More types of applications (e.g. not with just one input)

Thank you!

Source: external_pres.rst
48/48