VI. Distributed Algorithms

Goal: Simulate useful properties of centralistic systems in more or less poorly ordered distributed systems.

c.f. I-29

[Shingal et al. 1994]: *A distributed system consists of autonomous computers without any shared memory and without a global clock. Computers communicate using message-passing on a communication network with arbitrary delays.* ♦

$\Longrightarrow$ **Suitable techniques do not use 'global knowledge'**

Essential: Assumptions w.r.t system model *should be clear!*

- Communication style:
  - point-to-point between single processes
  - broadcast to *all* processes (of a process group)
- Communication guarantees: lost messages, message order, ...
- Which node failures are acceptable for an algorithm?

Reason: *Algorithms do not work without these assumptions!*

Overview: Basic Distributed Algorithms

1. Time and Causality
2. Applications of logical time to message ordering
3. Applications of Time to Distributed Mutual Exclusion and Fairness
4. Consistent global snapshots and checkpointing
5. Determination of 'global' system states: Termination, Deadlocks
6. Distributed Coordination: Leader Election

$\Longrightarrow$ **Characteristic techniques for solving distributed problems**

Note: *There are many algorithms/aspects we do not discuss here!*

$\ast$ Byzantine Agreement details $\Longrightarrow$ **MSc literature**
$\ast$ 2/3 Phase Commit protocols/transactions $\Longrightarrow$ **MSc literature**

$\ast$ Algorithms dedicated to unstructured Peer-to-Peer systems
$\ast$ Distributed Ledger Algorithms, Bitcoins, Etherium, ...

VI.1 Time and Causality

Properties of 'real' Time:

linear order: total order relation

Past (linear)/Present/Future (branching)

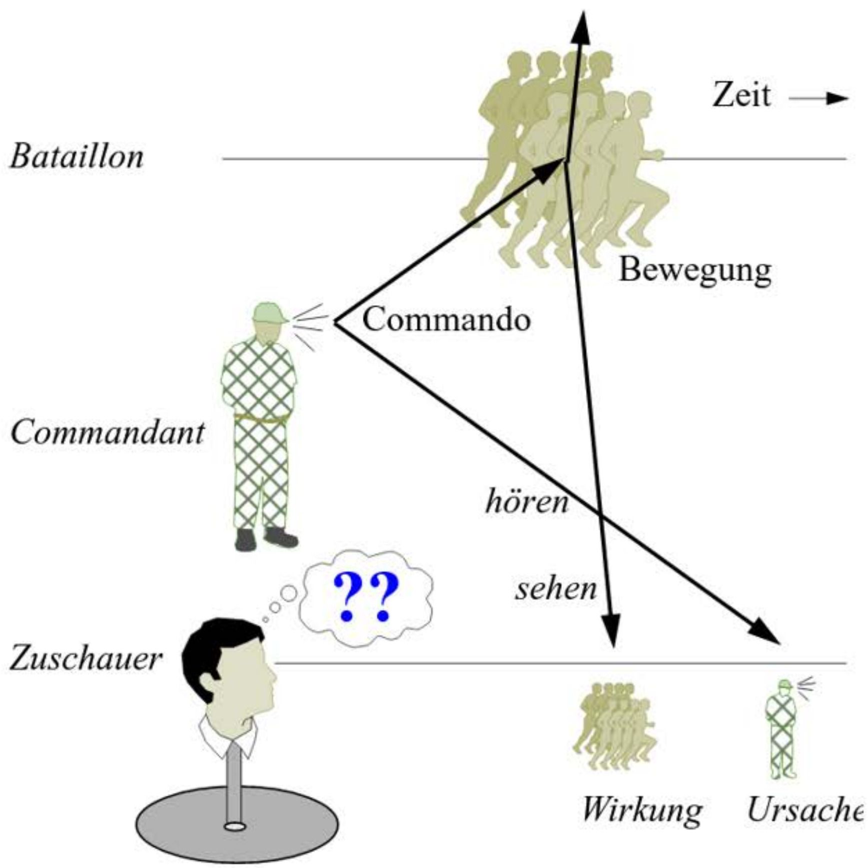
continuous; dense $\Longrightarrow$ **real** numbers as suitable basic model?

Fact: 'Full real time' is not always required

- ▶ traffic lights; backups based on time of day, ... (points in time)
- ▶ Accounting of resource usage (period of time)
- ◀ Decision criteria and priorities for:
  - Resource allocation, e.g., CPU scheduling
  - Election algorithms ('Eldest') VI.6.1
- ◀ Version control: text; source-code, e.g., build-organization
- ◀ Synchronization: a_2 *before* a_1; not *at the same time*

$\Longrightarrow$ **Typically not all properties of real time are really needed.**

Example: Observations and Causality



Wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so sieht er übereinstimmende Bewegungen desselben plötzlich eintreten, *ehe* er die Commandostimme oder das Hornsignal hört; aber aus seiner Kenntnis der *Causalzusammenhänge* weiß er, daß die Bewegungen die *Wirkung* des gehörten Commandos sind, dieses also jenen *objectiv* vorangehen muß, und er wird sich sofort der Täuschung bewußt, die in der Umkehrung der Zeitfolge in seinen Perceptionen liegt.

Christoph von Sigwart (1830-1904) *Logik* (1889)

Observer: Far away on a hill looking down on military exercising suddenly sees soldiers starting to run and shortly **after** that he also hears the commander who shouts 'RUN'

➔ **Effect is observable before the cause?**

$\Longrightarrow$ Observer time line contradicts rule of cause and effect

1. $P_{commander}$ sends $\texttt{msg}_1$ to soldiers (PS)
2. PS reacts, e.g. by running, which is observed by P_{observ} as $\texttt{msg}_2$
3. P_{observ} hears command $\texttt{msg}_1$ $\Longrightarrow$ Effect observed before cause?

Rule of Causality: Cause $\longrightarrow$ Effect

◁ Observer observes effect before cause

◁ **Alibi principle**: Speed of light and the impossibility to be at two distant places at the same time

◁ Messages: send is always before the corresponding receive

◁ Time paradox in 'backward' time travel
Example: Kill the inventor of the time machine. . . ?

Note: internal relative order is important, not relation to real time.

$\Longrightarrow$ **Simple linear orders are sufficient for most CS applications:**

▷ Position in FIFO-queue simulates relative arrival time

▷ 'Age' of a process simulated by strict monotonous numbering

▷ Version control for programs based on ordered Dewey-Notation

▷ Mutual exclusion and fairness based on request ordering

$\Longrightarrow$ **always use the most efficient but sufficient model**

Computer Systems: Real vs. Logical Time

1. **'Real' Time:** *Reference to external 'world' and time*
   Example: systems that control machines or traffic lights
   Physical Clock $\approx$ acceptable deviation from 'real time'
   - internal physical clocks (quartz crystal oscillation)
   - clock alignment within local/global networks
   - external points of reference, e.g. external time server NTP

   $\Longrightarrow$ costly in distributed systems if highly accurate

2. **Logical Time:** *internal, relative causality-based order*
   $\Longrightarrow$ reference to real time not needed
   Logical Clock $\approx$ internally consistent, no reference to real time
   - `integer` counter for logical steps (*ticks*)
   - compare and align in the case of message exchange
   - initial time is globally 0 via `reset`

   $\Longrightarrow$ cheap and efficient in almost all distributed systems

VI.1.1 Physical Clocks (= real time ?)

◀ **Astronomical Time:** $\frac{solar\ day}{24*60*60} = \frac{solar\ day}{86400}$ $\approx$ solar second
Earth/Sun rotation constant, but earth rotation slows down
$\Longrightarrow$ solar days/seconds become 'longer' $\Longrightarrow$ mean value GMT

◀ **Atomic time** (TAI): 01.01.1958
1 solar second = 9.192.631.770 Caesium 133 transitions

▶ **Universal Coordinated Time** (UTC): compensates for Drift
currently 3 millisec/day $\Longrightarrow$ TAI $\oplus$ leap seconds (approx. 0.9 sec) Savage CACM 09/2015 see vc
$\Longrightarrow$ lots of problems in IT due to 'repeated second'

▶ **Distribution:** accuracy of source $\oplus$ transfer time !

- Radio-based signals, e.g., DCF77 $\approx \pm$ 1 msec $\oplus$ $\pm$ 10 msec
- Satellites, e.g., GPS or GNSS (multiple satellites) 10^{-9} 10^{-6}
  $\approx$ 10 nanosec $\rightarrow$ 1 microsec

Caution: 1 nanosec $\approx$ 750 instructions in a 749.070 MIPS processor AMD/ Rizen 9 3950X (2019)

Effects of Errors: www.theregister.com/2016/02/03/decommissioned_satellite_software_knocks_out_gps/

Hardware Basis: Timer Chip

- Quartz ticks as source for time steps; counter decrement; reset;
- Software clock: clock ticks counted via interrupts based on counter
- Deviation $\approx$ 1 second in approx. $11\frac{1}{2}$ days
  clocks diverge in opposite 'direction' $\Longrightarrow$ approx. 2 seconds max

Algorithms: consistent initialization ($How?$)
keep deviation below threshold by msg exchange

Problems: (in all algorithms)

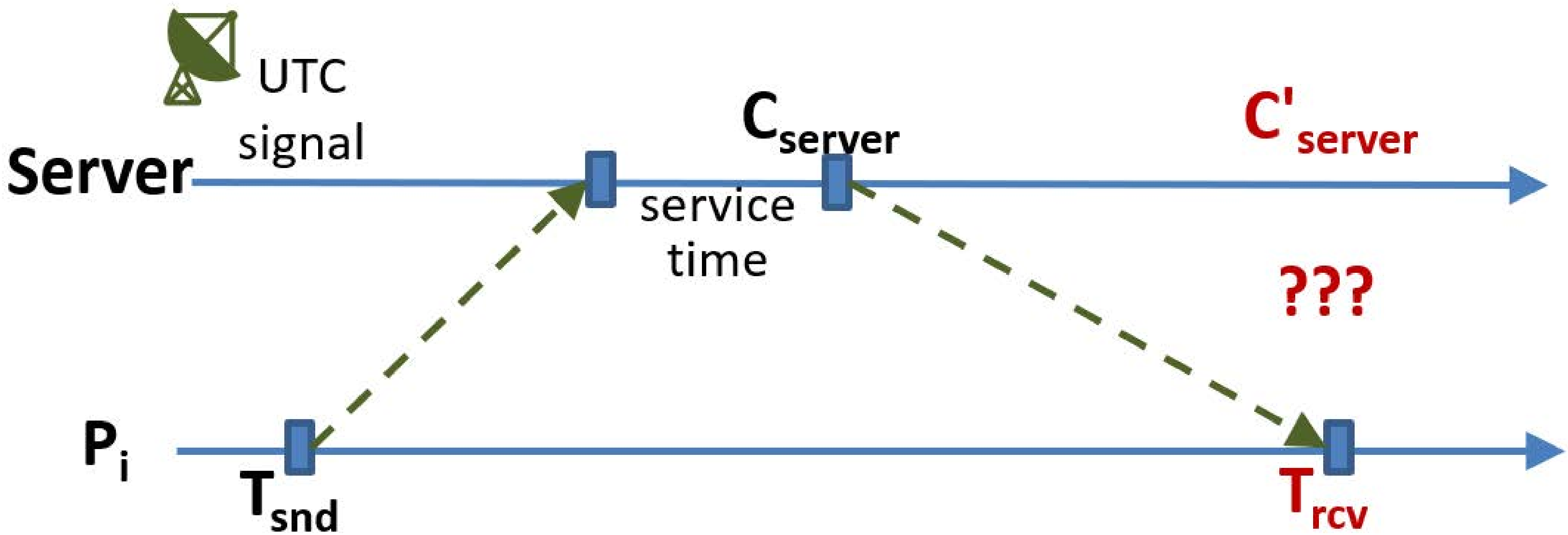
◀ **Transfer time:** Source $\longrightarrow$ Destination
different routes, number of hops, load levels etc.
$\Longrightarrow$ measure transfer times and use them in calculations

▶ **Suitable methods for adjustment**:
never put clock back; small adjustment instead of abrupt change
$\Longrightarrow$ slowdown/accelerate counter for local clock
Expl.: 100 IR/sec $\Longrightarrow$ 9/11 ms instead of 10 ms as compensation

Approximation of real time – 1

Christ-ian 1989

1. **Passive Time Server**
   (a) Time-Server holds 'external real' time in clock C
   (b) P_i sends Request at T_{snd} and receives C_{server} at T_{rcv}
   (c) $C^{rcv}_{server} \approx C_{server} + \frac{T_{rcv} - T_{snd}}{2}$ (plus service time on server)



◀ Same time for both messages? (e.g. uni-directional ring)
◀ Server and network may have varying loads

Approximation of real time– 2

2. **Active Time Server**, e.g., `Berkeley UNIX`
   (a) initially: correct setting of server clock
   (b) **Protocol:** in fixed intervals
      i. Server sends T_{server} to all (local) network nodes
      ii. $\forall P_i \in PS$: compute and send $\Delta_i := T_i - T_{server}$ to server by respecting transfer times
      iii. mean value of divergences is used to correct local clocks

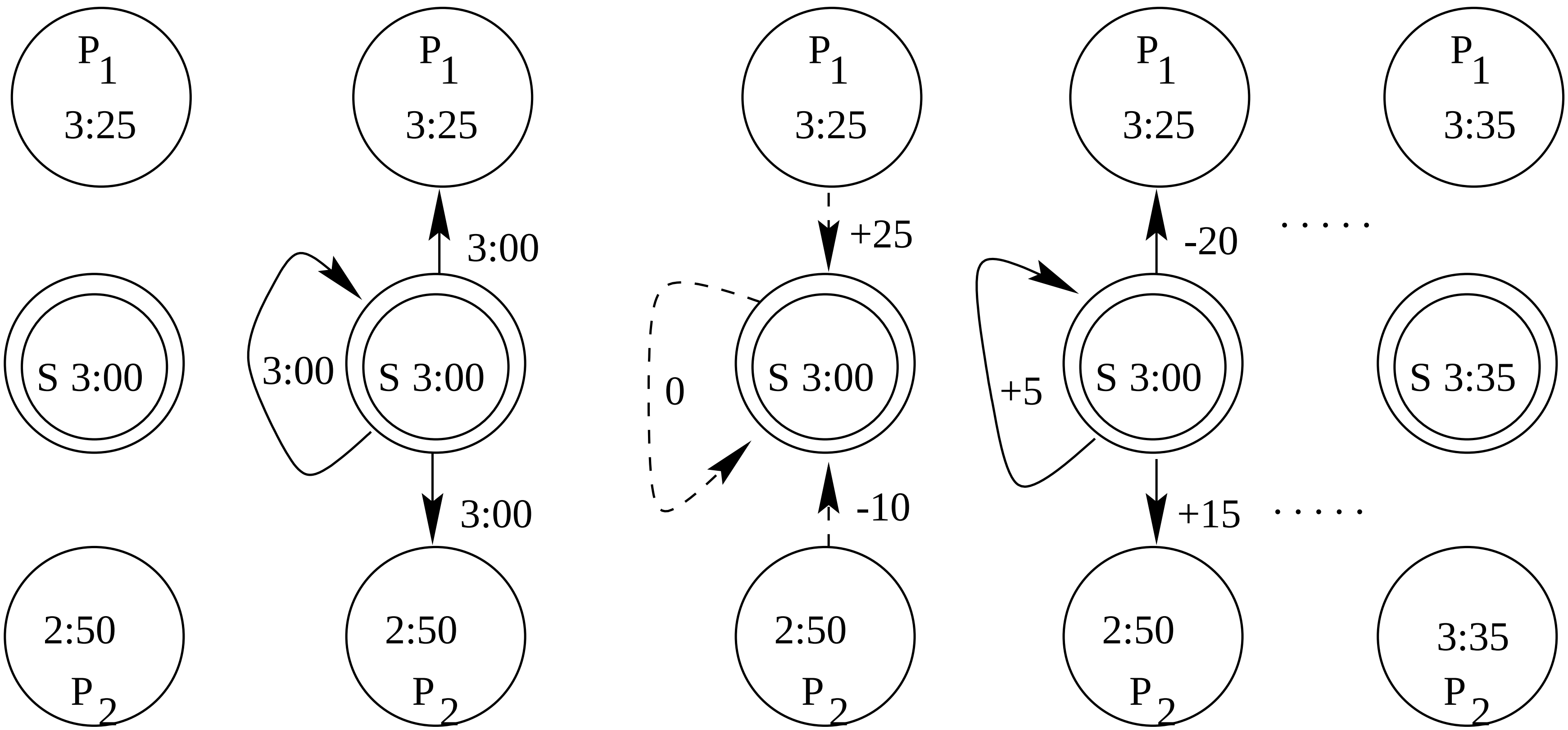
Gusella et al. 1989

`man timed` c.f. pg. VI-11

3. **Distributed Adjustment** (active)
   - Re-Synchronize in fixed (local) intervals
   - broadcast local time for calibration to all nodes
   - compute local mean values for correcting the local clock requires minimal number of answers; ignores extreme outliers

4. **Multiple external clock sources** (in different nodes)

Approximation of real time – 3: active Timeserver

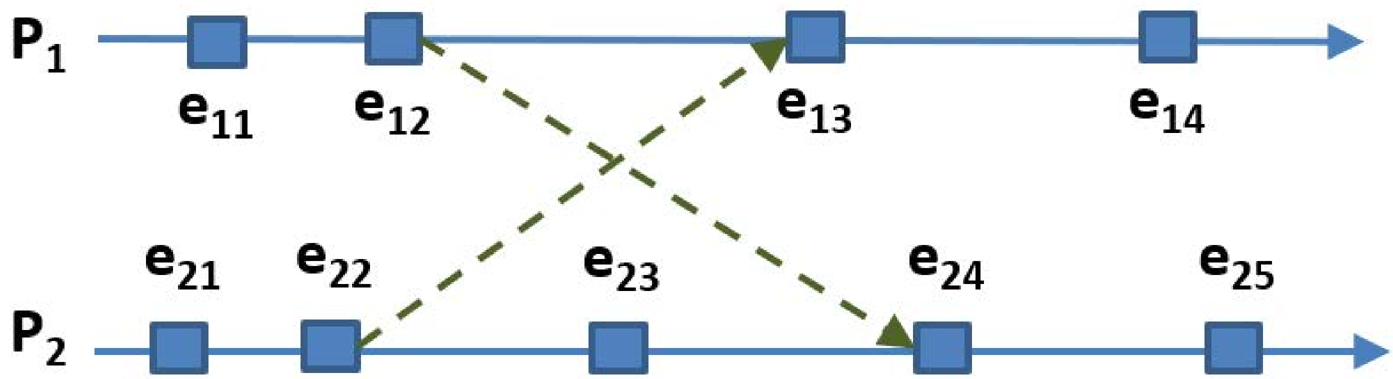


Algorithm: compute 'correct' time via arithmetic mean
send computed deviations
correction via slowdown/acceleration of local counter

VI.1.2 Logical Clocks - Virtual Time

Cause – Effect relation among

- ▷ Actions or **Events** of the same process
- ▷ Events of different processes due to sending/receiving messages
- ▷ Transitivity



Induced order: (**causality** relation)

c.f. III.1

- **local:** $e11 \xrightarrow{\sqsubset} e12 \xrightarrow{\sqsubset} e13 \xrightarrow{\sqsubset} e14$
  and $e21 \xrightarrow{\sqsubset} e22 \xrightarrow{\sqsubset} e23 \xrightarrow{\sqsubset} e24 \xrightarrow{\sqsubset} e25$
- **Communication:** $e12 \xrightarrow{\sqsubset} e24$ and $e22 \xrightarrow{\sqsubset} e13$
- **Transitivity**, e.g., $e21 \xrightarrow{\sqsubset} e14$
- **concurrent**, e.g., $(e11, e21)$, $(e23, e12)$

Definition VI.1: (happened-before Relation)

Let a, b, c be events of a set of events E and P_i, P_j processes.

The relation $a \xrightarrow{\sqsubset} b$ holds $:\Longleftrightarrow$

1. $a, b \in P_i$ *and* $a \sqsubset b$ *in* P_i, *or* *(sequential order)*
2. $a \approx \texttt{snd(msg,}P_j\texttt{)}$ *in* P_i *and* $b \approx \texttt{rcv(msg,}P_i\texttt{)}$ *in* P_j, *or*
3. $a \xrightarrow{\sqsubset} c$ *and* $c \xrightarrow{\sqsubset} b \implies a \xrightarrow{\sqsubset} b$

The events a and b are

- *in a* **causality relation** $:\Longleftrightarrow (a \xrightarrow{\sqsubset} b) \vee (b \xrightarrow{\sqsubset} a)$
- **concurrent** $:\Longleftrightarrow \neg(a \xrightarrow{\sqsubset} b) \wedge \neg(b \xrightarrow{\sqsubset} a)$

♦

P and P' **without** communication $\implies$ all events are concurrent

▶ no problem for program logic as there is no interaction

◀ no global time among all processes of PS achievable

Lamport's logical clocks

CACM 21(7), 1978

Idea: global time $C : (E, \xrightarrow{\sqsubset}) \longrightarrow (\mathbb{N}_0, <)$ for entire PS
respects $\xrightarrow{\sqsubset}$ **without** extra **messages**

- $\forall\ P_i \in PS$ exists a **local counter** C_i initial 0
- $\forall\ a \in P_i$ exists a local, unique **time stamp** $C_i(a)$
  derived from local clock value C_i when executing action a in P_i

Two Rules for global time C ...

C1: $\forall\ P_i \in PS$: $a \sqsubset_{PS} b$ local in $P_i \implies C_i(a) < C_i(b)$
Time respects local, internal order in each single process

C2: $\forall\ (a_{snd}, b_{rcv})$ where $a_{snd} \approx$ `snd(msg,`P_j`)` in P_i and
$b_{rcv} \approx$ `rcv(msg,`P_i`)` in P_j
$\implies C_i(a) < C_j(b)$
Time respects causality between corresponding send/receive

... **ensures:** $a \xrightarrow{\sqsubset} b \implies C(a) < C(b)$

Lamport Clocks – Implementation of C1 and C2

IR1: Increment local clock C_i **before** each new action

$\forall\ P_i \in PS\ \forall\ a \in P_i$ assign $C_i := C_i + d$ where $(d > 0)$
before executing action a
i.e. $a \sqsubset_{PS} b \implies C_i(b) = C_i(a) + d \implies C_i(a) < C_i(b)$

IR2: Propagate local time information with each message

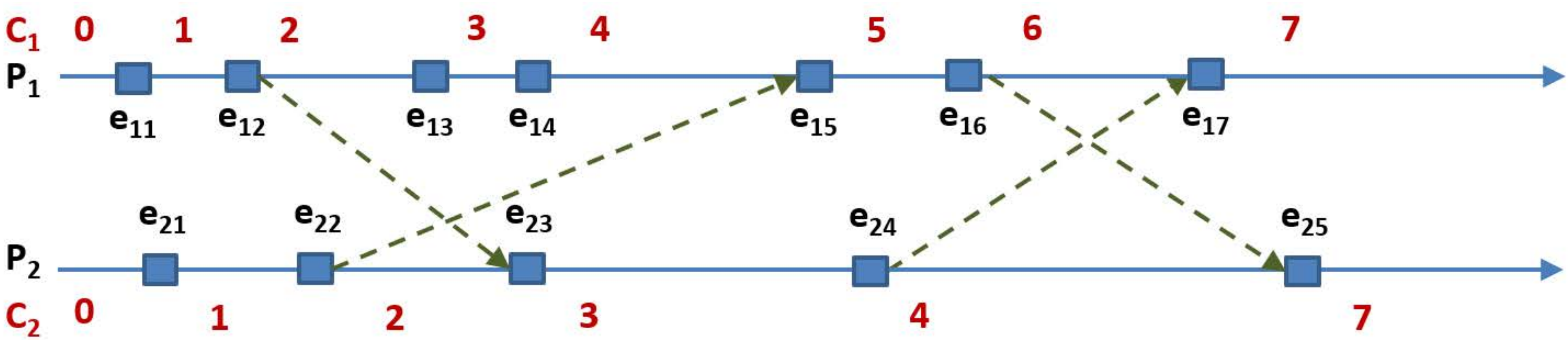
- Let $a_{snd} \approx$ `snd(msg,`P_j`)` in P_i
  1. C_i is incremented locally to $C_i(a)$ in P_i
  2. a_{snd} is **extended** by `snd(msg,`t_{msg}`,`P_j`)` where $t_{msg} = C_i(a)$
- Let $b_{rcv} \approx$ `rcv(msg,`t_{msg}`,`P_i`)` in P_j
  1. C_j is incremented locally to C_{temp} in P_j
  2. $C_j :=$ MAX(C_{temp},$t_{msg} + d$) where $d > 0$ (transfer time)

  i.e. $C_i(a_{snd}) = t_{msg} < t_{msg} + d \leq C_j(b_{rcv})$

Lamport Clocks – Example

▷ $e12 \xrightarrow{\sqsubset} e23$: $Max(\overbrace{2+1}^{rcv}, \overbrace{2+1}^{t_{msg}+d}) = 3$

▷ $e16 \xrightarrow{\sqsubset} e25$: $Max(\underbrace{4+1}_{rcv}, \underbrace{6+1}_{t_{msg}+d}) = 7$



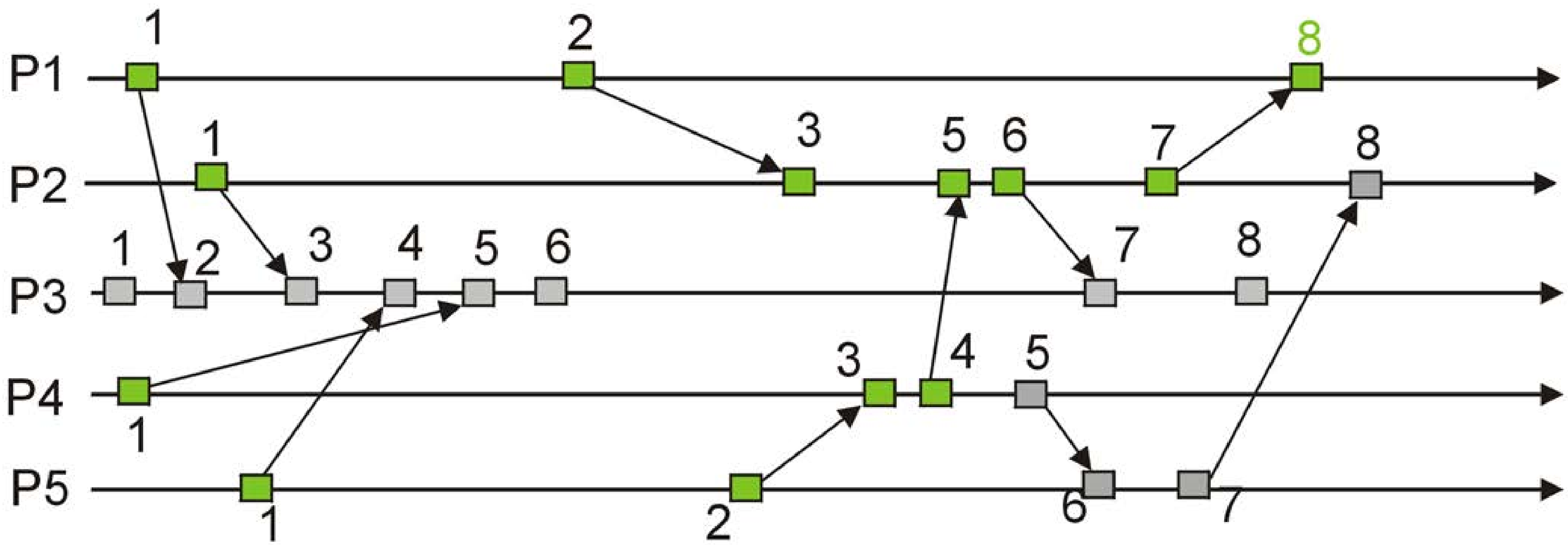
Observations:

◀ result is a partial order, e.g., (e_{14}, e_{23}) are concurrent

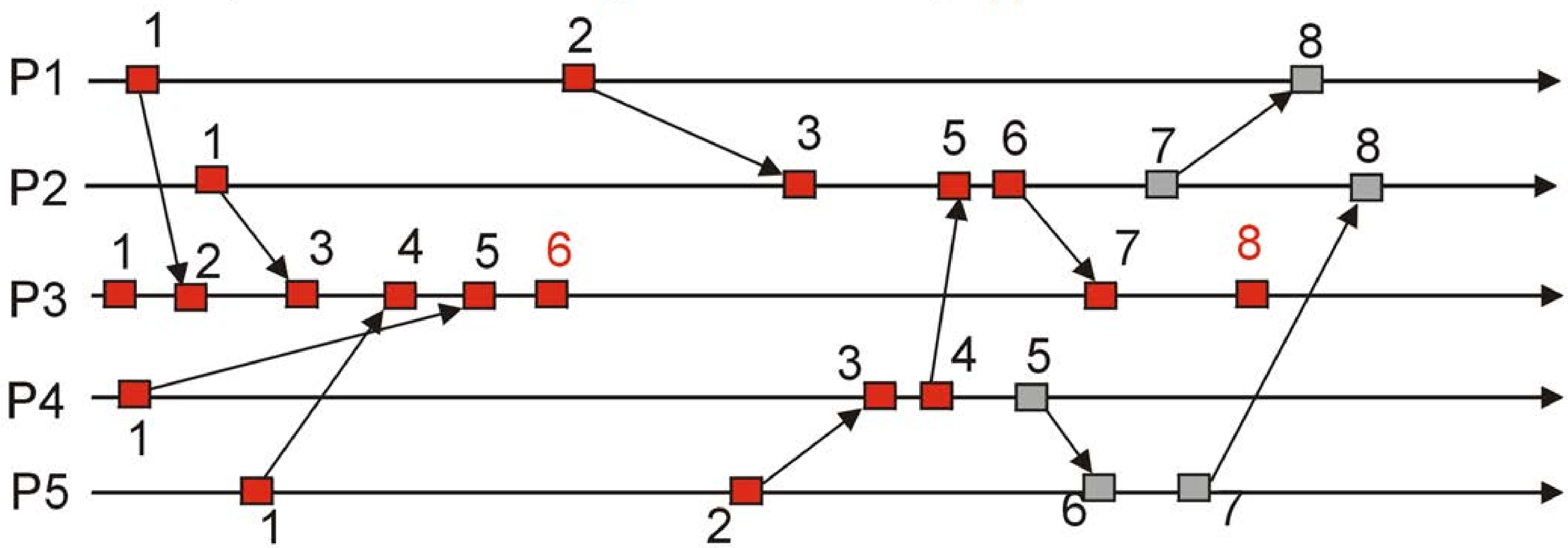
◀ $P_i \neq P_j$ use same clock value for different events, e.g., e_{17}/e_{25}

◀ concurrent events may have different clock values
Example: $(e23, e16)$ concurrent, but $C(e23) = 3 < C(e16) = 6$

Expl.: 'History' of selected Events in the same PS



Two ‚chains‘ of causal dependencies w.r.t. ‚happened-before‘



How to define a total order among events?

◀ Mapping C: $(E, \xrightarrow{\sqsubset})$ onto $(\mathbb{N}_0, <)$ is not *injective*

▶ Based on a unique numbering scheme for all processes $|\mathsf{PS}|$:
$a \xrightarrow{total} b \;:\Longleftrightarrow C_i(a) < C_j(b)$ or $(C_i(a) = C_j(b)) \wedge i < j$

▶ new mapping from $(E, \xrightarrow{total})$ onto $(\mathbb{N}_0 \times \mathbb{N}_0, <)$ is injective

▶ lexicographical order based on Lamport time and process indices
$\Longrightarrow$ **mapping** $(E, \xrightarrow{total})$ **onto** $(\mathbb{N}_0, <)$ is injective!

Problem: C **does not uniquely denote causality!**

$$C(a) < C(b) \implies \neg(b \xrightarrow{\sqsubset} a) \qquad \text{(not } \Longleftarrow\text{)}$$

◀ Increment may be caused locally or by send/receive?

◀ Most recent information only w.r.t. sender and receiver

Loss of structural information: $\xrightarrow{\sqsubset} \mapsto <$ and $\xrightarrow{\sqsupset} \mapsto >$

but: $\parallel \; \mapsto \{<, =, >\}$

Extending Lamport Time to Vector Time

Mattern 1987 Fidge 1988

Idea: Propagate more detailed and also indirect information

- $|PS| = n \implies \forall\, P_i \in PS\ \overrightarrow{C_i} = < C_i[1], \ldots, C_i[n] >$
- $\forall\, P_i \in PS$ is $\overrightarrow{C_i}$ initialized by $\overrightarrow{0}$
- $(i \neq j) \implies \overrightarrow{C_i}$ and $\overrightarrow{C_j}$ almost always **different**
  - $C_i[i] \approx$ local time C_i in P_i
  - $C_i[j]$ where $(i \neq j) \approx$ **most recent** information in P_i about the local clock value of P_j
    - a is the most recent action in P_j where $a \xrightarrow{\sqsubset}^{*} b$ holds and
    - b is the most recent action in P_i

    $\implies C_i[j] = C_j(a) + \Delta$ where $\Delta > 0$
- **Update:** Messages in PS are extended by so-called **time vectors**
  $\implies$ only moderate additional overhead w.r.t. Lamport time
  - Information dissemination much faster
  - Processes **without** interaction are not ordered (as before)

Implementing C1 and C2 using Vectors:

c.f. pg. VI-14

IR1: Increment the local clock C_i before each new action

$\forall\ P_i \in PS\ \forall\ a \in P_i$ assign $C_i[i] := C_i[i] + d$ where $(d > 0)$

$a \sqsubset_{PS} b \implies C_i[i](b) = C_i[i](a) + d \implies C_i[i](a) < C_i[i](b)$

IR2: Propagate time vector with each message

- Let $a_{snd} \approx$ `snd(msg,`P_j`)` in P_i
  1. $C_i[i]$ is incremented locally to $C_i[i](a)$ in P_i
  2. a_{snd} is extended by `snd(msg,`$\overrightarrow{t_{msg}}$`,`P_j`)` where $\overrightarrow{t_{msg}} = \overrightarrow{C_i}$
- Let $b_{rcv} \approx$ `rcv(msg,`$\overrightarrow{t_{msg}}$`,`P_i`)` in P_j
  1. $C_j[j]$ is incremented locally in P_j
  2. $\forall\ k \in [1:n]\quad C_j[k]$ **:= MAX(**$C_j[k]$,$t_{msg}[k]$**)**

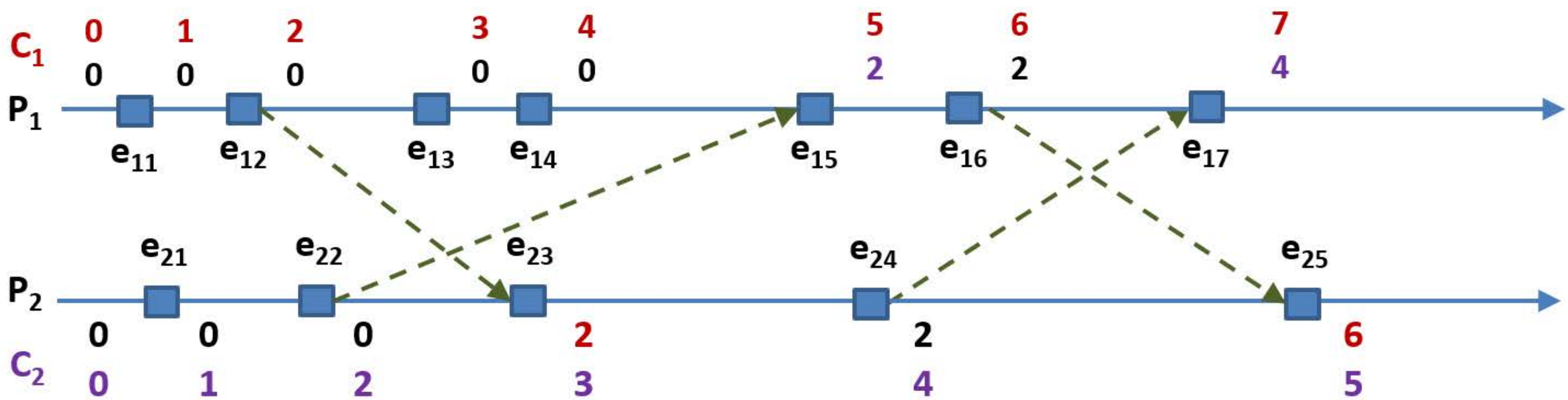
i.e. corresponding elements of vector $\overrightarrow{C_i}(a_{snd}) \leq \overrightarrow{C_j}(b_{rcv})$

Predicate: $\forall\ i \in [1:n]\ \forall\ j \in [1:n]$ holds $C_i[i] \geq C_j[i]$

local time in P_i always more recent than its approximation in P_j.

Example: Vector Clocks – 1

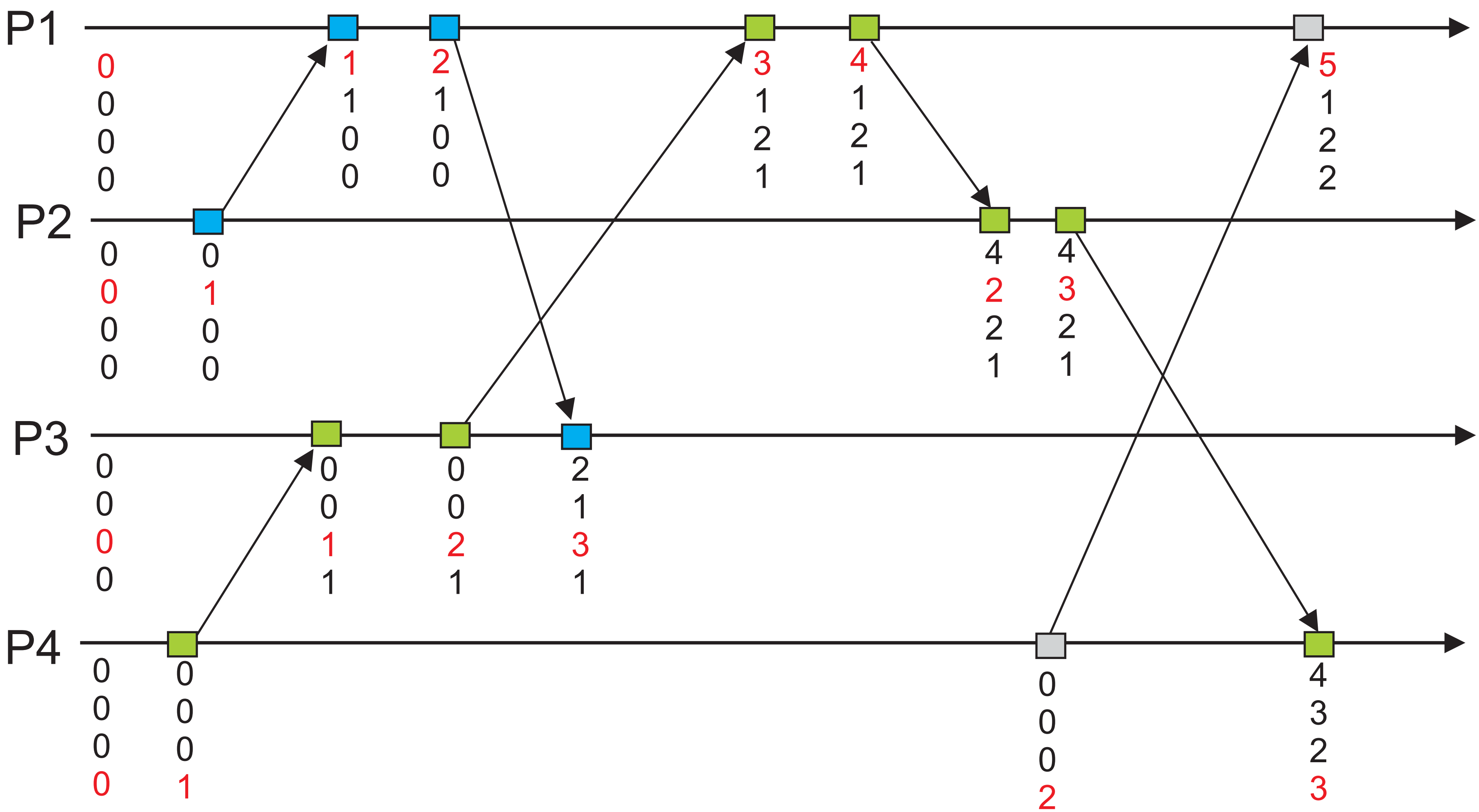
- $e16 \xrightarrow{\sqsubset} e25$ and $C(e16) = <6,2> \; < \; <6,5> = C(e25)$
- $(e23, e16)$ not ordered and $C(e23) = <2,3>$ concurrent $<6,2> = C(e16)$



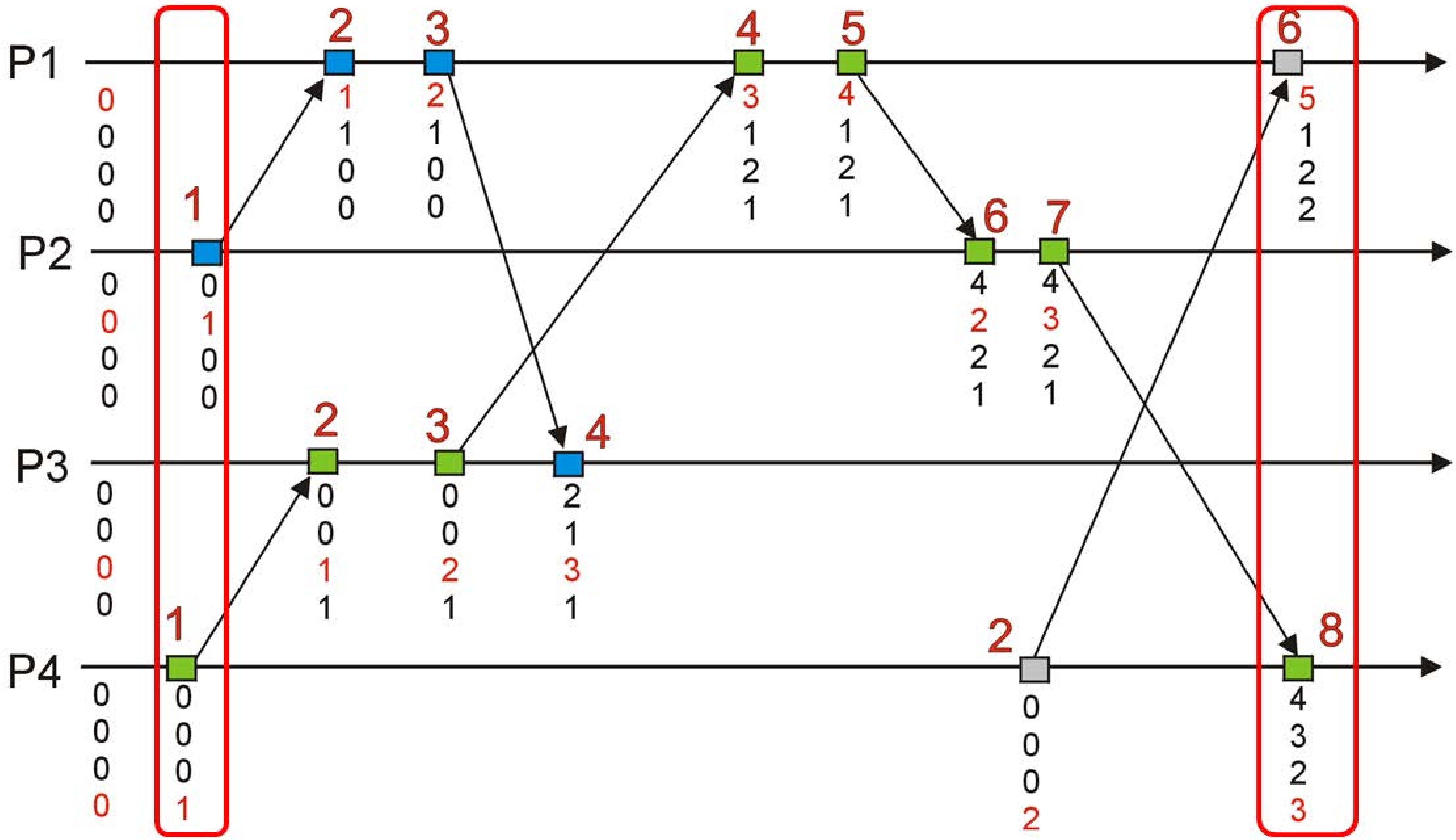
Comparison of vector clocks: (for actions a and b)

1. $t^a = t^b \; :\Longleftrightarrow \forall \; k \in [1:n] \;\; t^a[k] = t^b[k]$
2. $t^a \leq t^b \; :\Longleftrightarrow \forall \; k \in [1:n] \;\; t^a[k] \leq t^b[k]$
3. $t^a < t^b \; :\Longleftrightarrow (t^a \leq t^b) \wedge \neg(t^a = t^b)$
4. t^a **concurrent** $t^b \; :\Longleftrightarrow \neg(t^a < t^b) \wedge \neg(t^b < t^a)$

Example: Vector Clocks – 2



Example: Comparison of Lamport and Vector Time



Vector time: More information than Lamport

Advantage of vector clocks: $a \xrightarrow{\sqsubset} b \iff t^a < t^b$

$\implies$ **Causality can be derived from time-stamp alone!**

Reason:

- $t^a < t^b \implies \forall\, k \in [1:n]$ holds $t^a[k] \leq t^b[k]$ **and**
  $\exists\, l \in [1:n]$ where $t^a[l] < t^b[l]$
- $<$**-Entries:** local increment distinguishable from message
  $\implies$ **step-wise backtracking** of message transfers
  terminates (**finite** number of processes **and** a-cyclic chain)

Important: 'Logical time' definition depends on algorithm

- *Only events important to an algorithm are time-stamped.*
  e.g., R/W in mutual exclusion, msg snd/rcv in msg ordering, ...
- Size of counters and message overhead is reduced.

Applications of logical time models

c.f. VI.2/3

Origin: Lamport introduces 'logical time' for fairness in distributed mutual exclusion algorithms. (in: CACM 21(7), 1978)

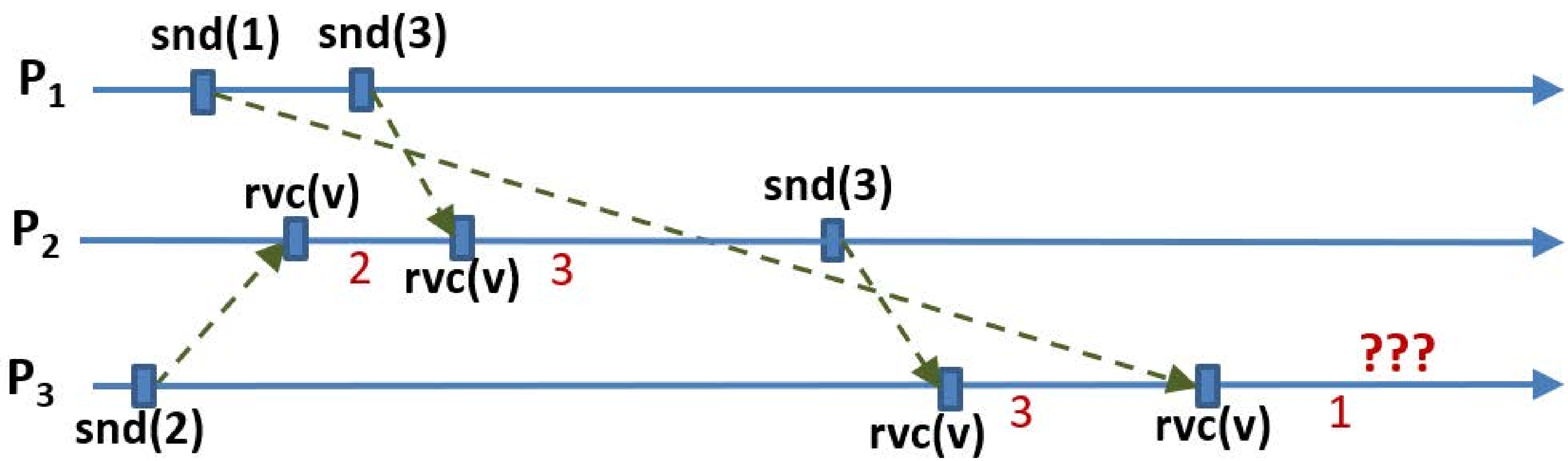
1. **Message ordering** for point-to-point and broadcast messaging VI.2
2. **Distributed mutual exclusion** VI.3
   - ◁ Request ordering for preventing deadlocks
   - ◁ Request ordering for ensuring fairness
3. Optimistic Concurrency Control in distributed database systems
   - conflicts based on mutual dependencies
   - optimistic $\approx$ conflicts are assumed to be rare
   - in case of conflict: choose 'victim' process and reset process
   - transactions are time-stamped for victim selection
4. . . .

End of VI.1

VI.2 Message Ordering

Problem: messages do not arrive in the order they were sent

- Effect: race conditions as in write-write conflicts
  Example: data base updates; the last update 'wins'
- Program logic based on causality may fail



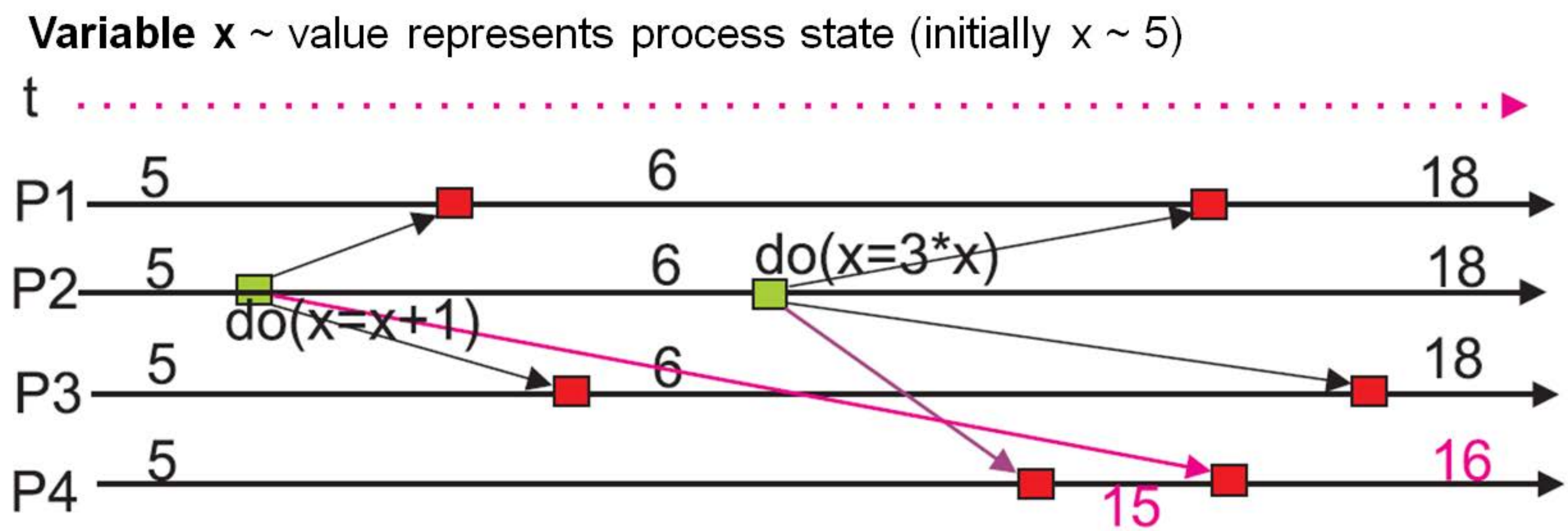
Recall: Isolated ordering between exactly two processes is simple
in the context of lossless message transfer.

III-65

Ordering of Broadcast-Messages

- **FIFO broadcast:** All Broadcast-Msg of a **single** sender P_{snd} arrive for all receivers P_{rcv} in the order as sent.

  Example: Problems with a broadcast without FIFO semantics



- **Causal broadcast:** P_{rcv} accepts a broadcast msg from P_{snd} **only after all** messages accepted in P_{snd} **before sending** this broadcast msg are also received and accepted in P_{rcv}.

Note: Causal broadcast $\Longrightarrow$ FIFO broadcast

Causal Broadcast: Birman/Schiper/Stephenson

1991

Preconditions

1. Broadcast message transfer is lossless
2. Each $P_i \in PS$ uses a vector for counting message send events

Idea: Time stamps VT for all broadcast send actions of all processes

1. Increment $VT_i[i]$ and send updated vector as part of message t_{msg}
2. Use MAX function to update local vector VT_i based on t_{msg}

Test in P_{rcv}: *Compare local* VT_i *to time stamp vector* t_{msg}

- If **all** local VT_i entries are equal or higher than those in t_{msg}
  - $\Longrightarrow$ all messages known in P_{snd} have also been accepted in P_{rcv}
  - $\Longrightarrow$ **accept** incoming message at once
- If VT_i **misses** messages $\Longrightarrow$ **buffer** arriving message in P_{rcv}
  until local vector VT_i is up to date w.r.t. t_{msg}, i.e., P_{snd}

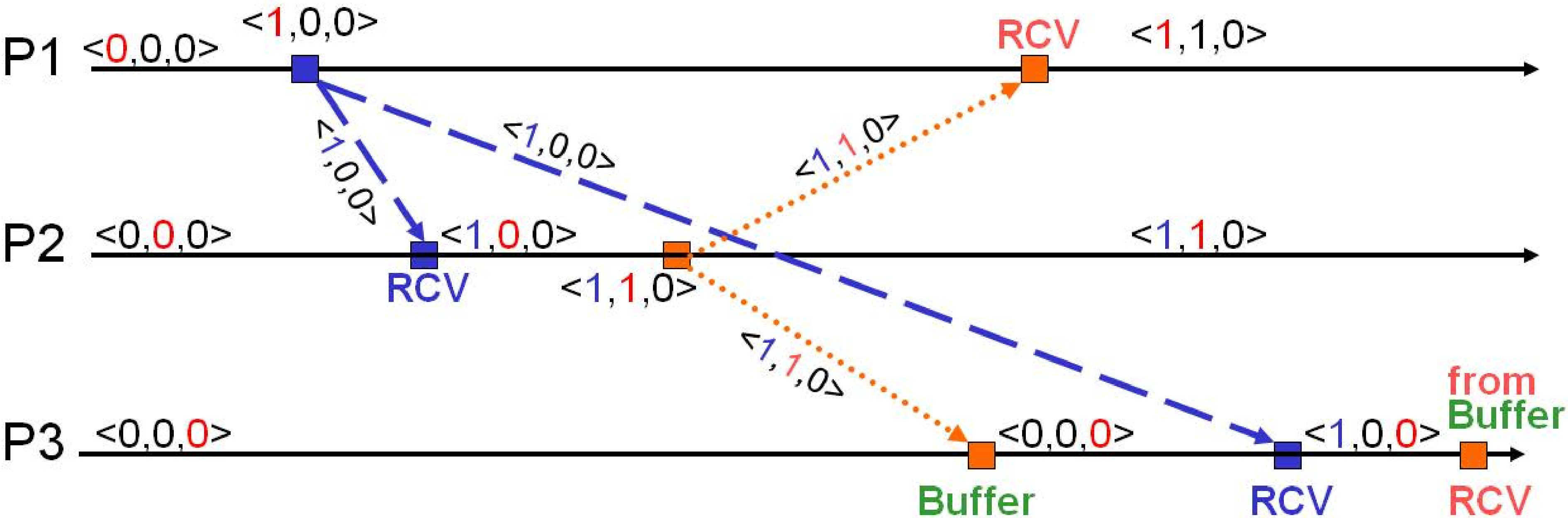
Note: logical time 'counts' broadcast events only

Birman/Schiper/Stephenson-Protocol

1. $\forall\ P_i \in PS$ initialize $\overrightarrow{VT_i}$ by $\overrightarrow{0}$
2. **Before** a broadcast in P_i $\approx$ increment $VT_i[i]$ locally
   $\Longrightarrow VT_i[i] = |$Messages from $P_i|$
   and send $\overrightarrow{VT_i}$ as part $(\overrightarrow{t_{msg}})$ of the original message
3. **All** P_j where $(i \neq j)$ receive message including $\overrightarrow{t_{msg}}$
   P_j **buffers** message **until:**
   (a) $VT_j[i] = t_{msg}[i] - 1$ $\Longrightarrow$ message is most recent one from P_i
   and
   (b) $\forall\ k \in [1:n] \setminus \{i\}$ holds: $VT_j[k] \geq t_{msg}[k]$
   $\Longrightarrow P_j$ knows at least all messages known by sender P_i at the time of sending.
4. **Accept**: Increment $VT_j[i]$ and execute **Buffer-TEST** (3.a/b)

Note: logical time $\approx$ Number of messages sent
acyclic time stamps $\Longrightarrow$ no Deadlocks possible in (3.b)

Example 'Run': Birman/Schiper/Stephenson



RCV in P2:
- <0,0,0>[1] = <1,0,0>[1] – 1
- Other entries in P2 >= Msg items

Buffer in P3: **<1,1,0>**
- <0,0,0>[2] = <1,1,0>[2] – 1
- BUT: <0,0,0>[1] < <1,1,0>[1]

  i.e. Msg from P1 not known in P3

RCV in P3:
- <0,0,0>[1] = <1,0,0>[1] – 1
- Other entries in P3 >= Msg items

RCV from Buffer: <1,1,0> ?
- <1,0,0>[2] = <1,1,0>[2] – 1
- **and** <1,0,0>[1] >= <1,1,0>[1]

afterwards: <1,1,0>

Note: `RCV/Buffer` events in P_2 and P_3 based on run

Point-to-Point Msg Ordering: Schiper/Egli/Sandoz 1989

Preconditions:

1. **Point-to-Point** message transfer is lossless
2. Each $P_i \in PS$ maintains vector time for all events

Idea: Send local state w.r.t. sending **to all other** processes
$\Longrightarrow$ information 'simulates' broadcast effect for receiving processes.

Data structure: $PS = \{P_1 \ldots, P_n\}$

- **local vector time** $\overrightarrow{V_i}$ for all processes $P_i \in PS$
- **Message-List:** Each $P_i \in PS$ uses ML_i to store pairs $(P_j, \overrightarrow{v_m})$
  holding knowledge about messages sent **to** P_j !!
  $\overrightarrow{v_m} \approx$ 'most recent' known vector time when sending in P_j
- **Messages:** (P_i, msg, $\overrightarrow{v_{msg}}$, ML_i, P_j)
  (P_{snd}, msg, P_{snd} time stamp, ML_i **without** current msg, P_{rcv})

Schiper/Eggli/Sandoz–Protocol – 1

1989

▶ **Sender** P_i**:** $\overrightarrow{V_i[i]} := \overrightarrow{V_i[i]}+1;$

`snd(`P_i`, msg,` $\overrightarrow{V_i}$`,` ML_i`,` P_j`);` Order !

ML_i `:= insert(`ML_i`,(`P_j`,` $\overrightarrow{V_i}$`));`

$\Longrightarrow$ current message **not** contained in ML_i message list

◀ **Receiver** P_j**:** on arrival of (P_i, msg, $\overrightarrow{v_{msg}}$, **ML**, P_j) (1)

`TEST: if` $\nexists$ $(P_j, \overrightarrow{v})$ `in` **ML** `then ACCEPT;` (2) $\in$ Msg

`else if` $\overrightarrow{v} \not\leq \overrightarrow{V_j}$ `then BUFFER(...);` (3)

`else ACCEPT;` `fi;` (4)

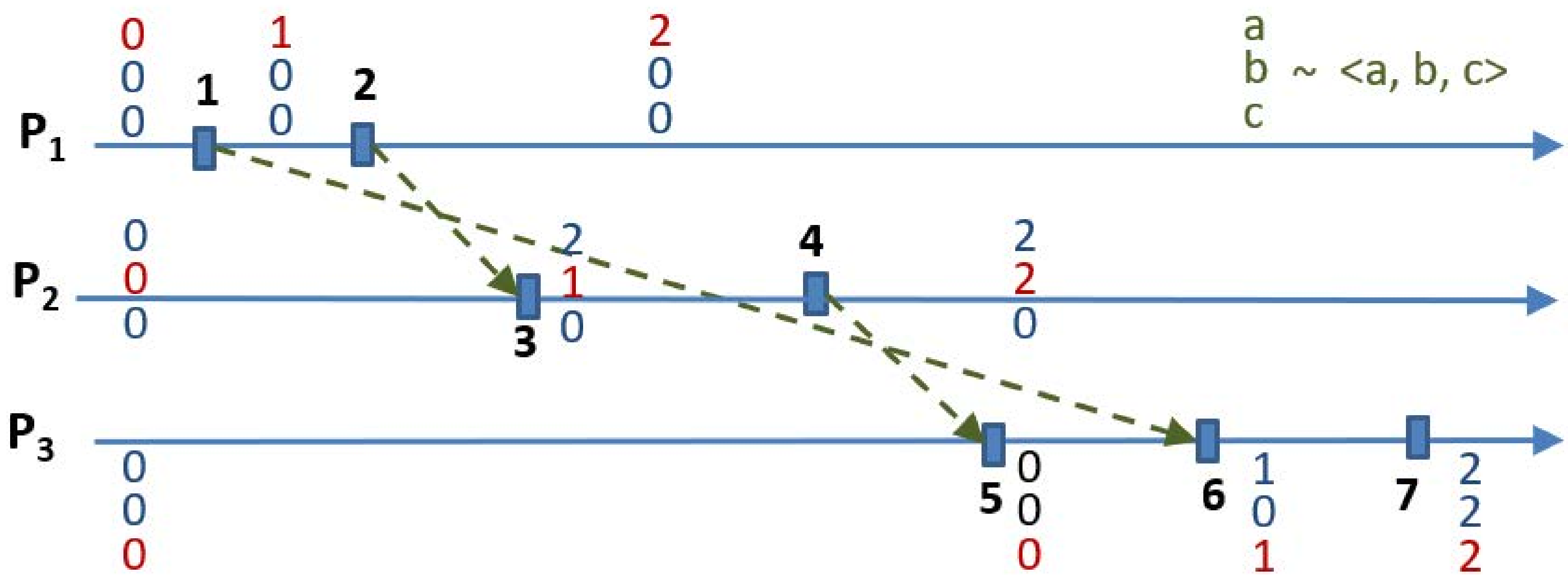
`fi;`

(2) $(P_j, \overrightarrow{v})$ in **ML** $\approx$ knowledge of P_i about messages to P_j

pair not found $\Longrightarrow$ **first message to** P_j (known in P_i)

(3) Condition holds $\Longrightarrow$ state in P_j is **not** more recent than knowledge in P_i w.r.t. msgs to P_j when sending $\Longrightarrow$ Buffer !

Schiper/Eggli/Sandoz–Protocol – 2



`ACCEPT`: a message $(P_i, \text{msg}, \overrightarrow{v_{msg}}, ML, P_j)$ from P_i in P_j

1. Combine $ML \oplus ML_j$ via insert/maximum for entries with $(k \neq j)$
   $\Longrightarrow$ only the most recent pair $(P_k, \overrightarrow{v})$ remains in ML_j
2. Increment local time $\overrightarrow{V_j}$ and compute maximum based on $\overrightarrow{v_{msg}}$
3. Repeat `TEST` for all messages buffered using new time stamp analogous to steps (2/3) (may result in consecutive `ACCEPT` steps)

Result: Causal Point-to-Point message ordering

Example – Schiper/Eggli/Sandoz–Protocol

c.f. pg. VI-33

1. **snd in P_1:** increment; `snd`(P_1, msg, $< 1, 0, 0 >$,$\emptyset$,P_3)
   afterwards: $ML_1 := [(P_3, < 1, 0, 0 >)]$
2. **snd in P_1:** increment; `snd`(P_1, msg, $< 2, 0, 0 >$,$[(P_3, < 1, 0, 0 >)]$,P_2)
   afterwards: $ML_1 := [(P_2, < 2, 0, 0 >), (P_3, < 1, 0, 0 >)]$
3. **rcv in P_2:** the message (P_1, msg, $< 2, 0, 0 >$,$[(P_3, < 1, 0, 0 >)]$,P_2)
   $(P_2, ?) \notin [(P_3, < 1, 0, 0 >)] \implies$ ACCEPT VI-32 (2)
   $ML_2 := [(P_3, < 1, 0, 0 >)]$; increment and MAX $\implies V_2 := < 2, 1, 0 >$
4. **snd in P_2:** increment; `snd`(P_2, msg, $< 2, 2, 0 >$,$[(P_3, < 1, 0, 0 >)]$,P_3)
   afterwards: $ML_2 := [(P_3, < 2, 2, 0 >)]$
5. **rcv in P_3:** the message (P_2, msg, $< 2, 2, 0 >$,$[(P_3, < 1, 0, 0 >)]$,P_3)
   $(P_3, < 1, 0, 0 >) \in ML$, but $< 1, 0, 0 > \not< < 0, 0, 0 > = V_3 \implies$ BUFFER VI-32 (3)
6. **rcv in P_3:** the message (P_1, msg, $< 1, 0, 0 >$,$\emptyset$,P_3)
   $(P_3, ?) \notin \emptyset \implies$ ACCEPT VI-32 (2)
   Combine $\emptyset$ and $\emptyset \implies ML_3$ remains empty; $V_3 := < 1, 0, 1 >$
7. **Buffer check in P_3** w.r.t. (P_2, msg, $< 2, 2, 0 >$,$[(P_3, < 1, 0, 0 >)]$,P_3)
   $(P_3, < 1, 0, 0 >) \in$ ML, but $< 1, 0, 0 > < < 1, 0, 1 > = V_3 \implies$ ACCEPT
   ML_3 remains $\emptyset$; $V_3 := < 2, 2, 2 >$

VI.3 Distributed Mutual Exclusion

Goals:

Def. III.8 III-25

- Safeness, i.e., correct critical section (csd) implementation
- Deadlock freedom, starvation freeness, fairness
- Efficiency: not too much overhead

Model: PS with Interaction using (asynchronous) message-passing

Costs:

▶ auxiliary data structures, vector clocks etc.

◀ additional processes and/or **messages**

Caution: single point-of-failure $\longrightarrow$ multiple point-of-failure ?

processes do not react, lost control messages . . .

$\Longrightarrow$ **only feasible under strong preconditions**

Logical Time: Fairness and avoiding cyclic waits (Deadlocks)

Approaches to Distributed Mutual Exclusion

- **Centralized approach:** Client/Server model, **but** server may be *bottleneck* $\Longrightarrow$ hierarchical systems, specialized server *single point of failure* $\Longrightarrow$ replicated, redundant servers

c.f. chapter III-66 ff.

- **Distributed approaches:** Permission to enter csd via . . .

III-71

. . . **Inquiry among processes:**

c.f. pg. VI-41

  - P_i uses message passing to get permission from other processes
  - Permission is considered granted iff majority accepts
  - Inform other processes about own requests and answer requests

. . . **Exclusive ownership of a control token**

  - Wait or ask for `Token` using message passing
  - Handover of token after usage or via answering token request

Variants: different pre-assumed topologies for message or token exchange with varying overhead etc.

Token-based Algorithms

Basic Idea: Access to `csd` is granted to $P \iff P$ owns **token**

Variants: Organization of token circulation through PS
underlying communication structure of process system PS

Precondition: secure message transfer without loss of control token !!

1. **Simple Algorithm:** PS organized as a logical **ring** $P \mapsto P_{next}$

   `rcv(token)` $\Longrightarrow$

   `IF (request(self)) THEN csd ELSE snd(`P_{next}`,token); FI;`

   Problems: 'unused' token circulates around the ring
   node crashes; lost messages

2. **Suzuki-Kasami Broadcast Algorithm** (1985)
   - ▷ P_i wants $csd \Longrightarrow P_i$ sends broadcast with REQ for token
   - ▷ Owner of token reacts only on incoming REQ
   - ▷ REQ messages are prioritized based on logical REQ counter

     $\Longrightarrow$ Token handover after csd acts fair based on time stamps

Suzuki-Kasami Broadcast Algorithm

Data structures: (REQ-counter as vector time)

- local $RN_i[i]$ counter for most recent request REQ in P_i
- Vector $RN_i[1 : |PS|]$ $\approx$ most recent, known REQ of other P_j
- **Token: Queue** Q holding **requesting** processes **and**
  Vector $LN[1 : |PS|]$ holding numbers of **granted** REQ

Problems to be solved:

◀ **Outdated** REQ should not be answered
Example: All processes receive REQ from P_1 that is granted by P_2; afterwards, other processes should not hand over to P_1 again
$\Longrightarrow$ **granted** requests LN are part of the token

▶ **Decide for next process to hand over:** (Fairness)
Queue Q of all processes with pending REQ is part of the token
and updated after each `csd` access

Suzuki-Kasami Broadcast Algorithm

- **Request in P_i:**

```
RN_i[i] := RN_i[i]+1;                                        /* Prologue */
broadcast(i,REQ,RN_i[i]); rcv(Q,LN);    /* to all procs including P_i */
        csd                      /* blocking rcv: wait for token ... csd */
LN[i] := RN_i[i];                                   /* Start of Epilogue */
FORALL j ∈ [1 : n] DO                                /* Token-Q update */
   IF (RN_i[j]==LN[j] + 1 AND P_j ∉ Q) THEN Q.enqueue(P_j); FI;
                OD;
IF (Q.notempty()) THEN snd(Q.frontdequeue(),Q,LN);   FI;
```

- **Reactions in P_i:**

```
rcv(j,REQ,RNr)  ⟹                                /* always able to react */
    RN_i[j] := MAX(RN_i[j],RNr);                    /* update REQ counter */
    IF (token() AND (NOT in csd/Epilog) AND (RN_i[j] == LN[j] + 1)
    THEN snd(P_j,Q,LN);
    FI;                                      /* empty Q due to Epilogue ! */
```

Properties of Suzuki-Kasami Broadcast Algorithm

- **Safeness:** At any time, token is owned by at most one process
  entering `csd` only possible when owning token
- **Fairness:**
  $(n+1)$-th REQ enters Q only if n-th request has succeeded
  $\Longrightarrow$ **maximum** number of ($|PS|$-1) processes in **FIFO**-Queue Q;
  P hands token to next process after `csd` if there is a REQ in Q

◁ If P_i owns token and there are **no** REQ in Q
  $\Longrightarrow$ P_i may use `csd` repeatedly

- **Blocking:** Token is owned by P_i after a finite time (c.f. Fairness)
  but: crashes, lost messages, non-terminating `csd`s
- **Cost:** $0 \ldots |PS|$ messages (REQs and token)

Simulation of central knowledge (server):

Current owner of token acts as the server for the `csd`.

Non-Token Algorithms

Preconditions:

- global, unique time stamps for all messages
- message transfer respects message ordering !

Structure: each process $P_i \in PS$ 'knows' about it's process sets

1. **Request set** $R_i \subseteq PS$: **ask** for permission to enter csd
2. **Inform set** $I_i \subseteq PS$: **notify** if local state w.r.t. `csd` changes

Idea: minimize sets for each P_i

- R_i: ensure that at most one process is able to enter `csd`
- P_i is part of a sufficient number of inform sets I_j of other processes $P_j \implies P_i$ has a solid basis for it's own decisions
- in case of conflict: minimal time stamp $\implies$ highest priority

Variants: Size of R_i and I_i
Number and content of messages required

Lamport Algorithm

CACM 21(7), 1978

Remark: First application of logical time concepts

- Simulate 'global server' through **all** processes of PS
  - **Request set:** $\forall\ P_i \in PS$ choose $R_i = PS \setminus \{P_i\}$
  - **Inform set:** $\forall\ P_i \in PS$ choose $I_i = PS \setminus \{P_i\}$

  $\Longrightarrow$ **global message exchange in** PS
- **Data structures** in each P_i: **Priority-Queue** Q_i holding request pairs
  - $< t_j, j >$ lexicographically ordered by **time stamps** in Queue
  - process numbers and Lamport clocks $\approx clock()$

  $\Longrightarrow$ globally ordered time stamps t_i

  c.f. pg. VI-18
- **Message types:** $REQ \approx$ request

  $REPLY \approx$ acknowledgement for REQ

  $REL \approx$ release

Algorithm Prologue/Epilogue and Process Behavior

- **Request in P_i:** $t_i \approx$ `clock();` /* Prologue */

  `FORALL` $P_j \in R_i$ `DO snd(`P_j`,`$< REQ, t_i, i >$`); OD;` /* inform */

  Q_i`.enqueue(`$< t_i, i >$`);`

  `WAIT UNTIL` /* periodic test */

  `FORALL` $P_j \in R_i$ `rcv(`P_j`,`$< REPLY, t_j, j >$`);` (1)

  `AND (`Q_i`.front() ==` $< t_i, i >$`);` (2)

  csd$_i$**;**

  Q_i`.dequeue();` /* Epilogue */

  `FORALL` $P_j \in I_i$ `DO snd(`P_j`,`$< REL, t_i, i >$`); OD;`

- **Reactive behavior in P_i at all times**

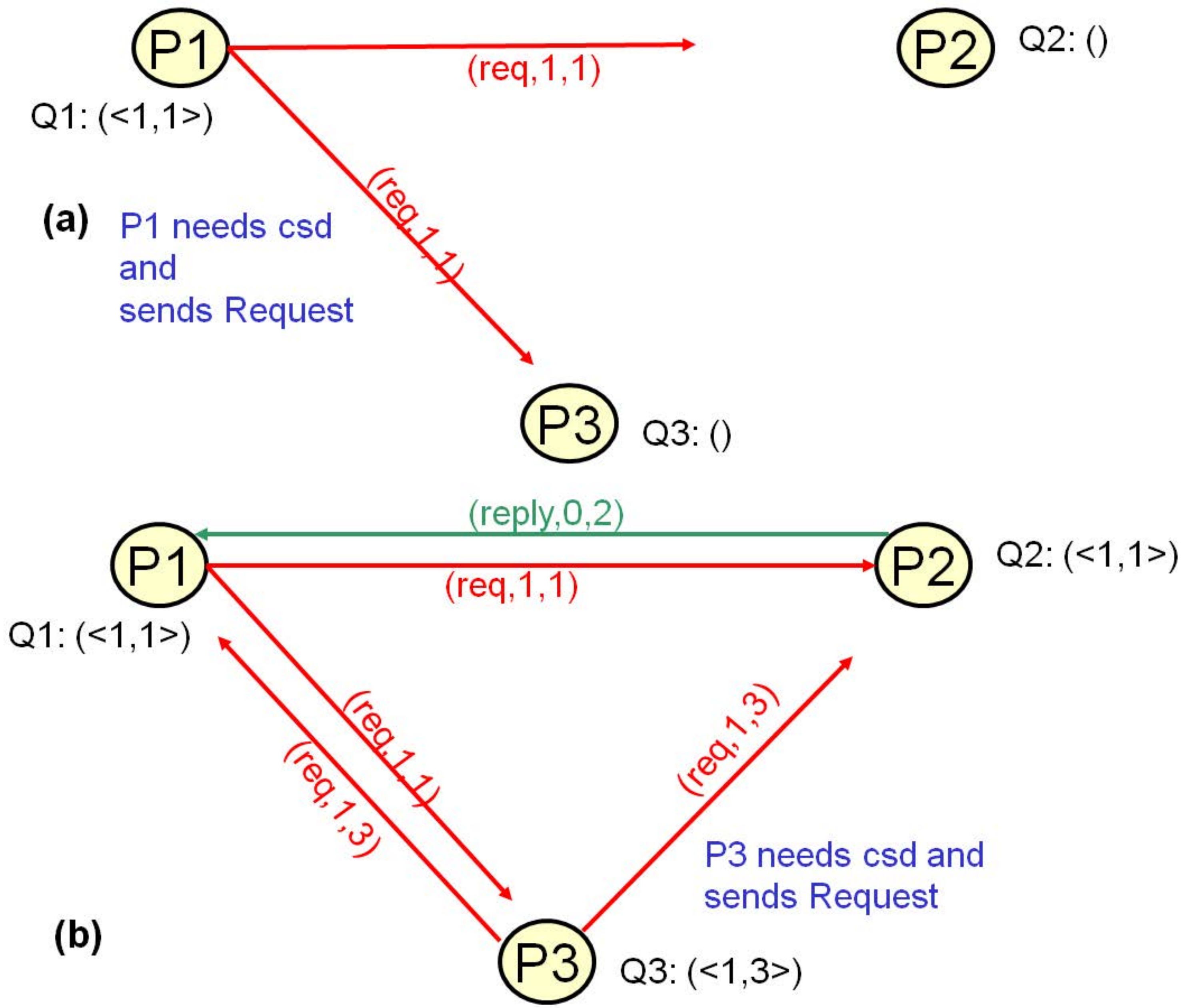
  `rcv(`P_j`,`$< REQ, t_j, j >$`)` $\Longrightarrow$ `snd(`P_j`,`$< REPLY, t_i, i >$`);` (3)

  Q_i`.enqueue(`$< t_j, j >$`);`

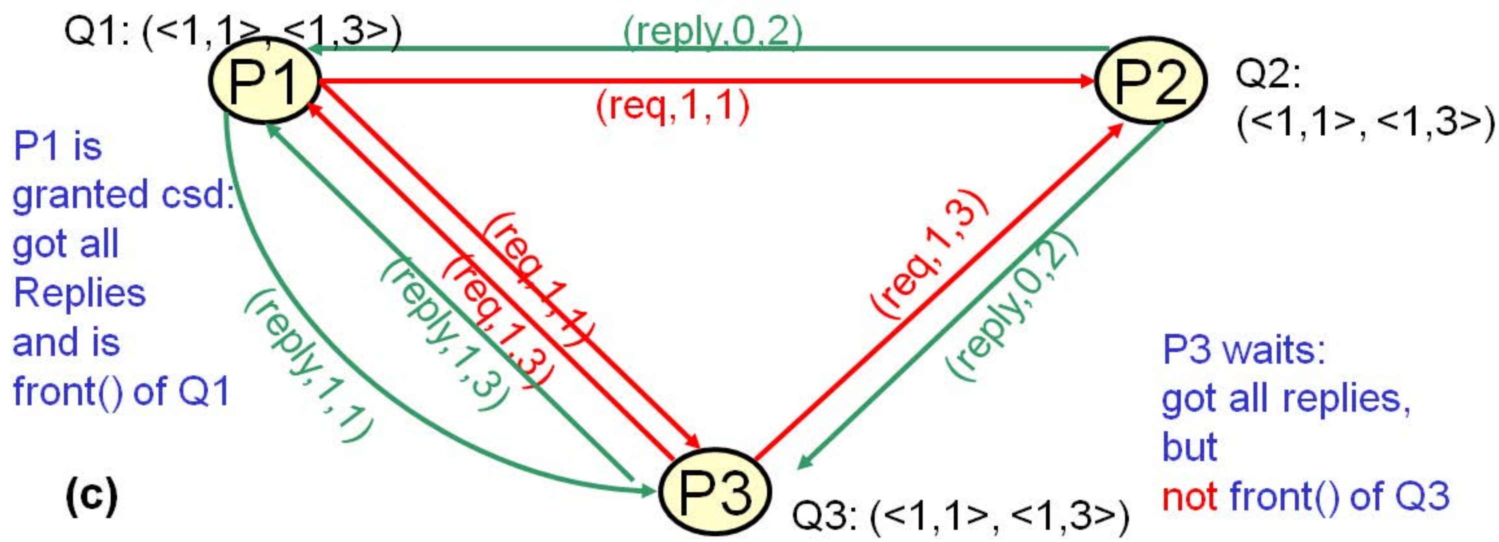
  `rcv(`P_j`,`$< REL, t_j, j >$`)` $\Longrightarrow$ Q_i`.remove(`$< t_j, j >$`);`

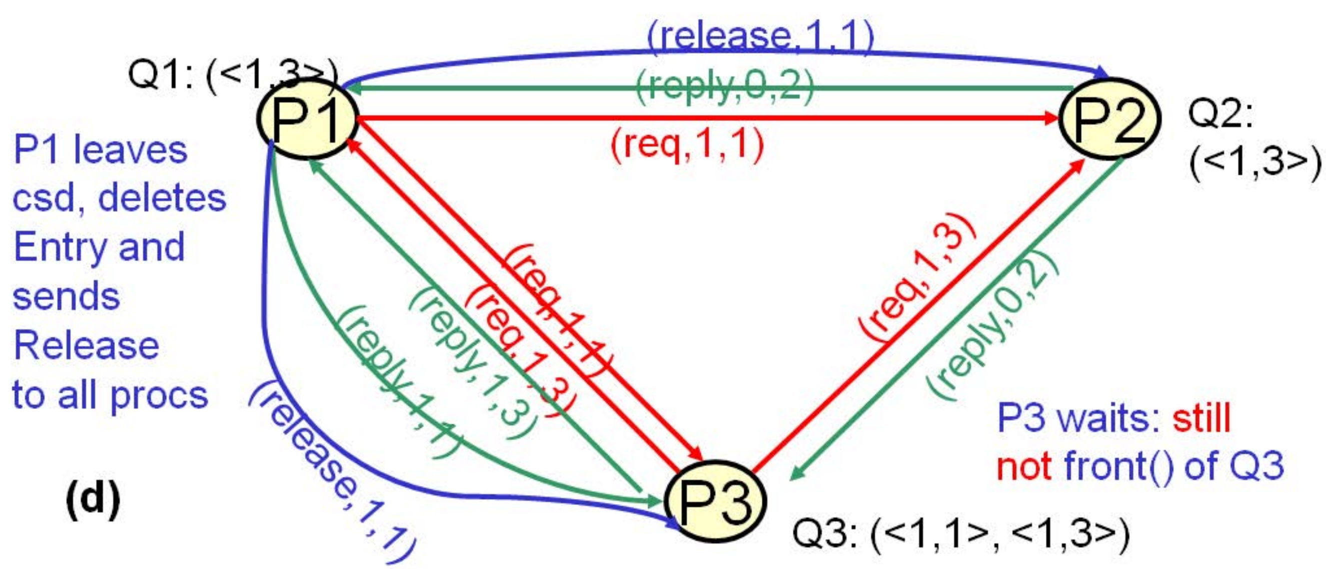
Example: Lamport Algorithm - Two Requests

Le-gend c.f. pg. VI-46

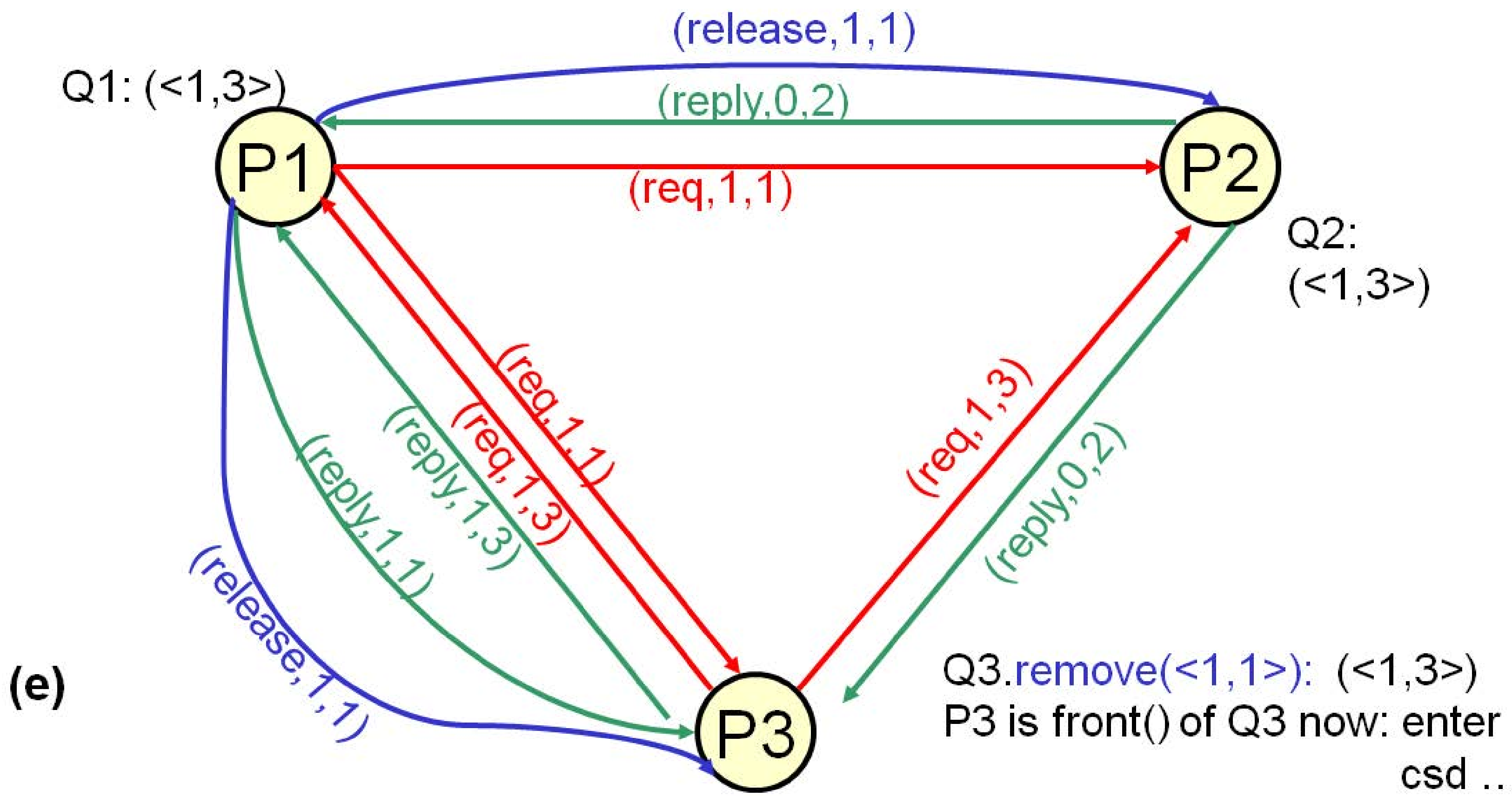


Example: Lamport Algorithm - Permission & Wait



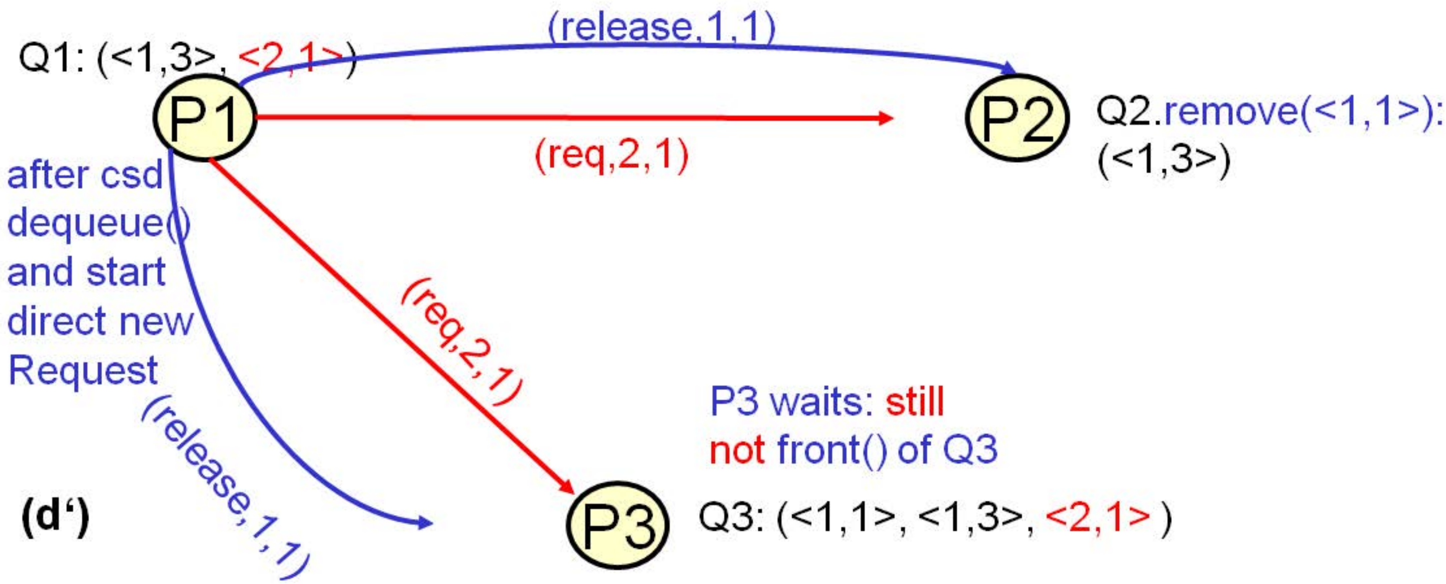


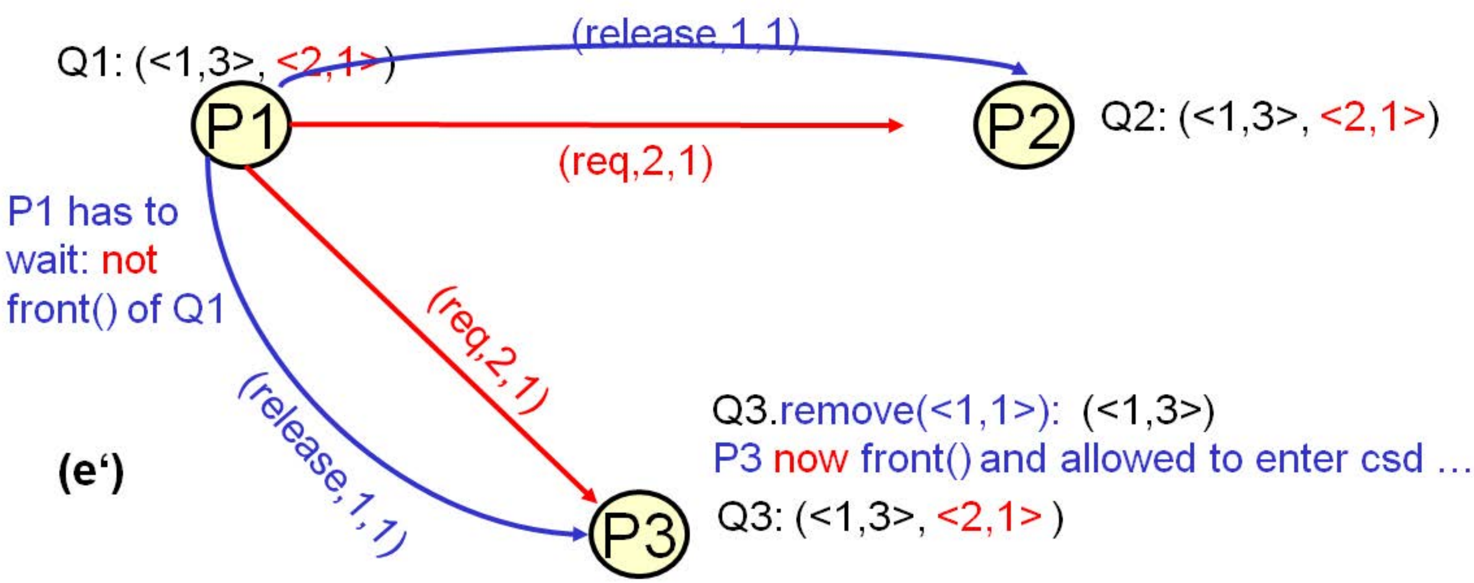
Example: Lamport Algorithm - Handover



Note: Request counter initial 0 in all processes;
before sending Request: increment by 1
All queues are empty initially
Messages carry 3 items of information (<Type>, ReqNr , ProcID)
where <Type> ::= req | reply | rel

Example: Lamport Algorithm - Unfairness ?





Assessment of Lamport Algorithm

- Safeness: P_i in $\texttt{csd}_i \implies$ Wait actions (1) and (2) executed

 (1) REQ; wait for $REPLY \implies$ processes know about REQ_i

 (2) own request is $\texttt{front} \implies$ other requests are more recent

 $\implies$ queues of all other processes wait for REL from P_i

- No permanent blocking: (3) ensures that all $REPLY$s are sent
- No deadlock: globally ordered time stamps prevent from cycles
- Fairness: conflicting requests are handled fair by REQ order queues

no lost msgs

- **Cost:** $3 * (|PS| - 1)$ message for each csd permission

 $\implies$ **optimization advisable**

- **Reliability:** *realistic preconditions?*
  - entire algorithm does not work if a single process fails to answer
  - lost $REQ/REPLY$ blocks P_i; lost REL blocks csd

 $\implies$ Use **time outs** to detect faulty processes

Optimization: Ricart-Agrawala Algorithm

CACM 1981

Basic Idea: combine $REPLY$ and REL messages

- **Request in P_i:**

FORALL $P_j \in R_i$ DO snd(P_j,$< REQ, t_i, i >$); OD; /* inform */

WAIT FORALL $P_j \in R_i$ rcv(P_j,$< REPLY, t_j, j >$);

csd$_i$;

FORALL $P_j \in Q_i$ DO snd(P_j,$< REPLY, t_i, i >$); /*Epilogue*/

$Q_i.remove$ OD;

- **Reactive behavior in P_i at all times:**

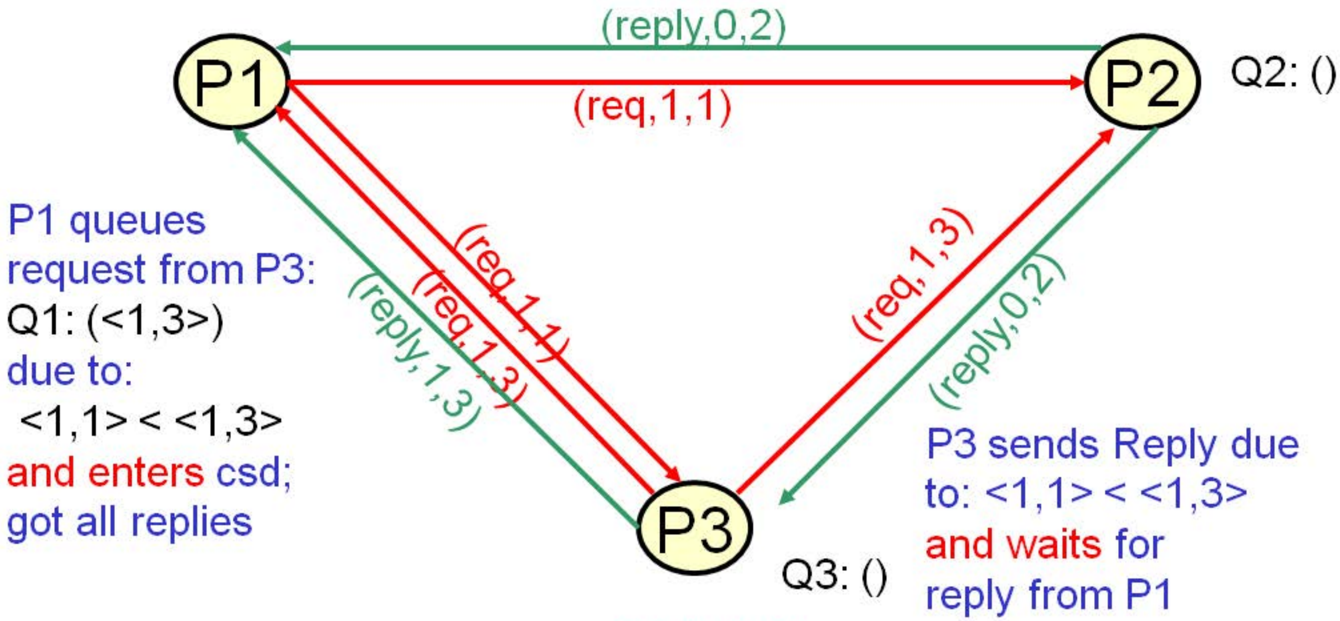
rcv(P_j,$< REQ, t_j, j >$) $\Longrightarrow$

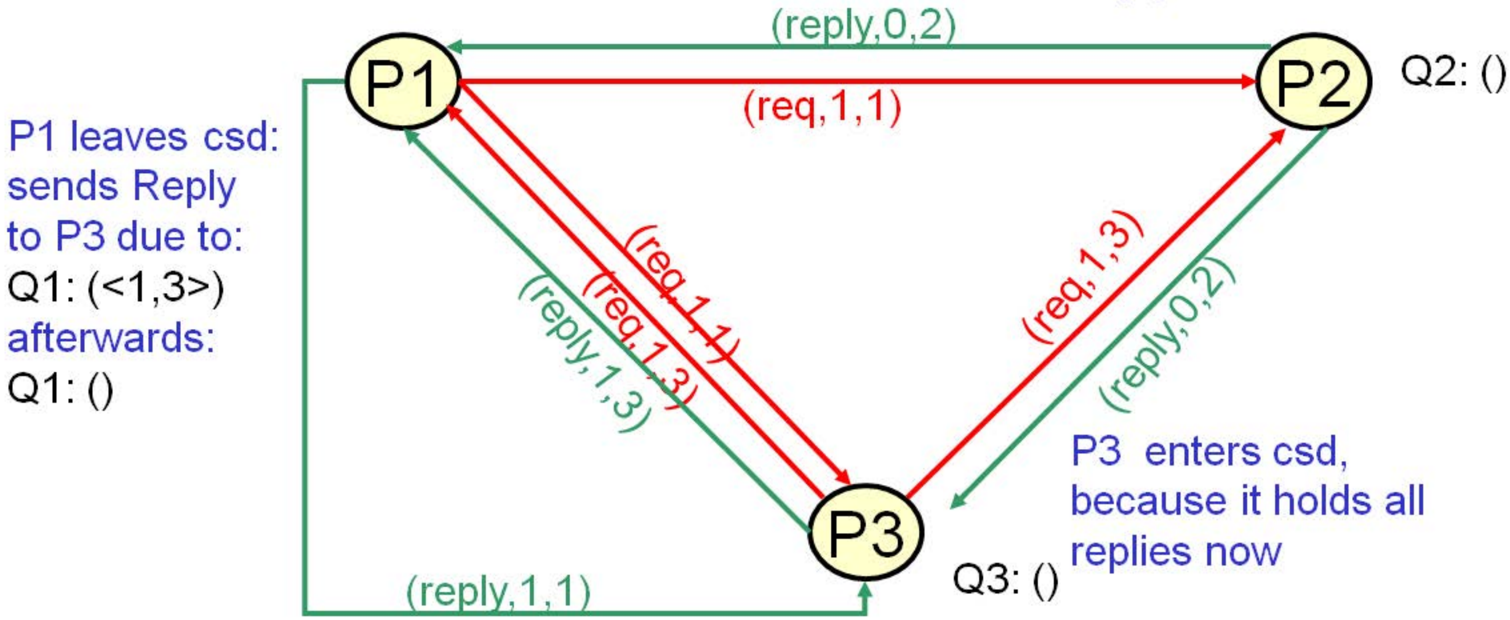
IF $[$ (P_i not in csd) AND ($t_j < t_i$) $]$ /* csd not needed */

THEN snd(P_j,$< REPLY, t_i, i >$); /* or 'older' request */

ELSE Q_i.enqueue($< t_j, j >$);

Example: Ricart-Agrawala Algorithm





Assessment of Ricart-Agrawala Algorithm

- Process stores requests only if it owns the csd
  - ▷ unburdens processes that don't need csd at all
  - ◁ not much redundancy for storing requests
- all processes have to react and send $REPLY$ messages
- when leaving $\texttt{cs}_i$ all pending `reply` messages are send

Safeness: P_i waits for **all** $REPLY$ messages (as before)
only processes P_j that are not inside the `csd`
or have only a lower priority request will answer

Cost: $2 * (|PS| - 1)$ messages for each `csd` permission

Note: *There are a lot more algorithms and optimizations dealing with distributed mutual exclusion. An overview can be found in*

P. C. Saxena and J. Rai: *A survey of permission-based distributed mutual exclusion algorithms.* Computer Standards & Interfaces, 25(2), 2003

VI.4 Global Snapshots and Consistency

Motivation: **Fault Tolerance**

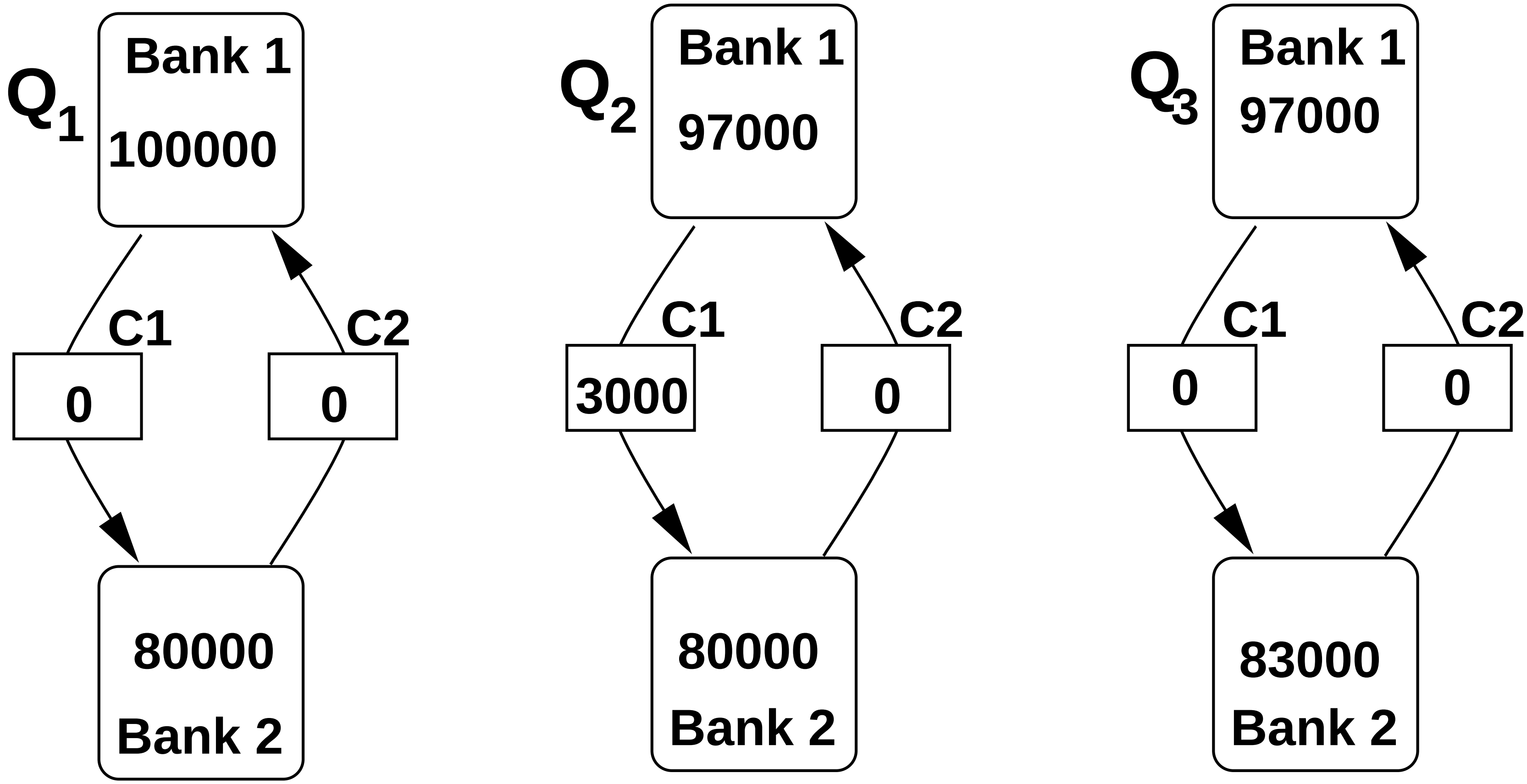
- **Why:** many compute nodes and complex networks
  $\Longrightarrow$ **high probability of (partial) failures**
- **How:**
  1. Store **local states** of processes
     Examples: MEM, Register, PC, Resources
  2. Store the content of **message channels**
  3. **Combine** distributed local states to global **Recovery Points**
  4. Crash $\Longrightarrow$ **Roll-back** based on most recent recovery point(s)

PCB cf. II-17

- **Problems:** additional overhead
  - Trade-Off: Memory/Communication vs. Rollback benefits
  - **Consistency** of stored 'global' states!

Overhead at runtime $\Longrightarrow$ Crash easier to handle

Example: Consistency is an Important Issue



- global state $\approx$ amount of money in (bank1, C1, C2, bank2)
- **inconsistent state recording without coordination, e.g.,**
  1. Bank1 in Q_1, channels/bank2 in Q_2: (100000,3000,0,80000) $\approx$ 183000
  2. Channels in Q_1, bank1/2 in Q_2: (97000,0,0,80000) $\approx$ 177000

Reasons for Consistency Problems

◀ **Processes:**

- uncoordinated recording of local states is not sufficient
- coordination based on 'global clock' usually not feasable

Combination: internal state **plus** view on external system

◀ **Message channels:** How to get content?

- record content of channels (before/after sending) **or**
- wait for channel clearance based on maximum transfer time

Important: (P_1, C_{12}, P_2) observe $|SND_{P_1}| = |C_{12}| + |RCV_{P_2}|$
without FIFO channels $\Longrightarrow$ explicit msg ordering required

Elementary Problem: $P_i \neq P_j$ record **different** external views

◁ wrong assumptions about messages sent

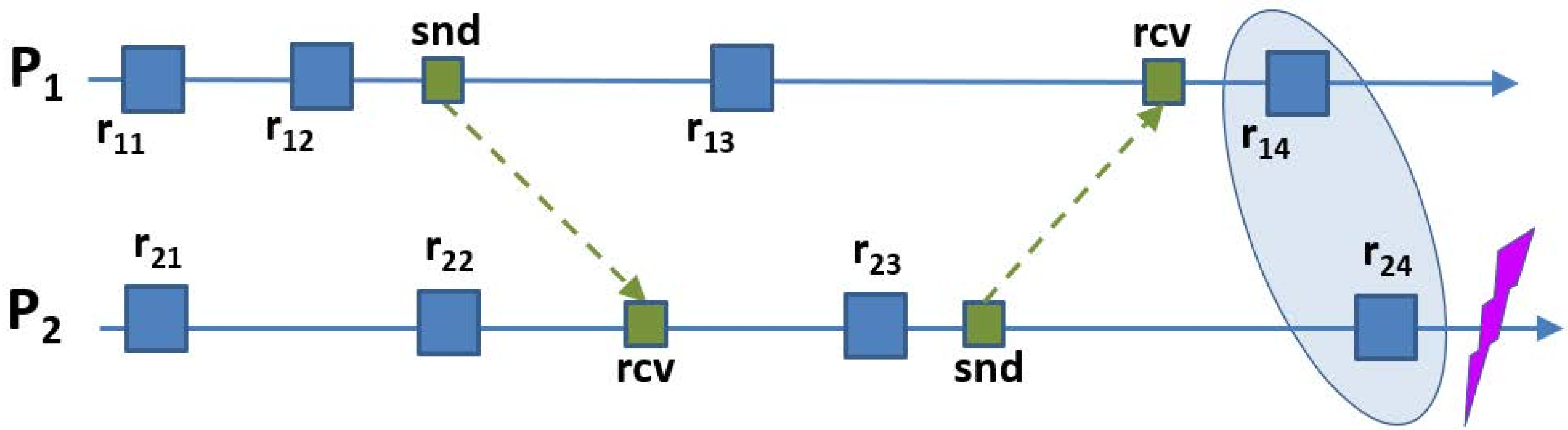
◁ wrong assumptions about messages received

◁ how to detect messages lost during transfer?

Problems using local Recovery Points only – 1

- Each process P_i records its own local recovery-points r_{ik}
- Recording is **not** coordinated among processes
- Crash $\Longrightarrow$ PS resets each P_i to it's *most recent* recovery-point
- Messages are re-send if sent after recording recovery-point

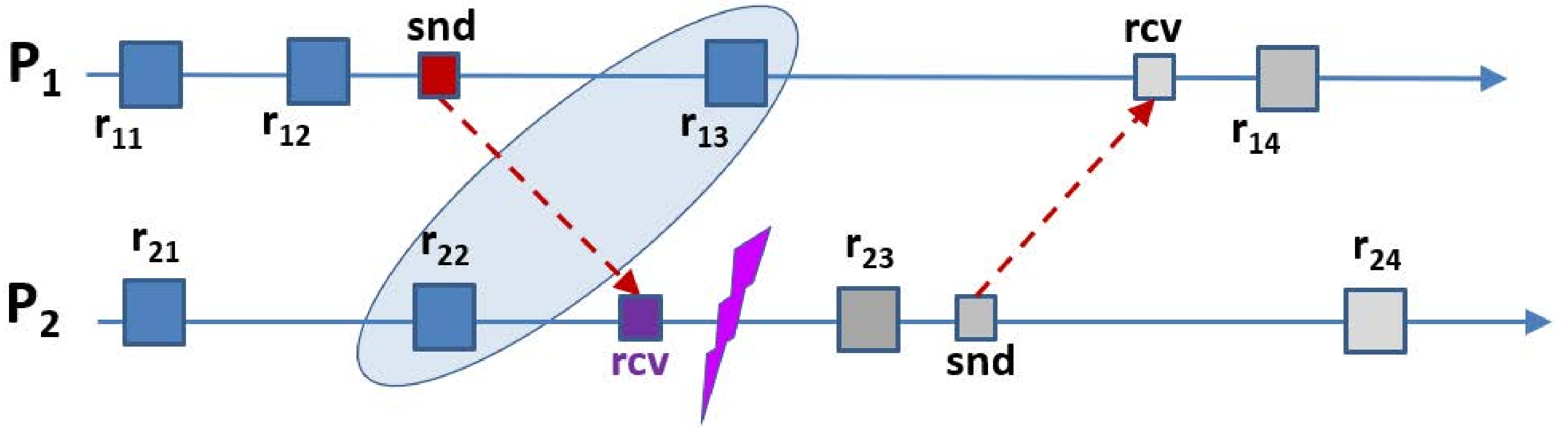
1. **Crash in P_2 $\Longrightarrow$ roll-back to state** (r_{14}, r_{24})



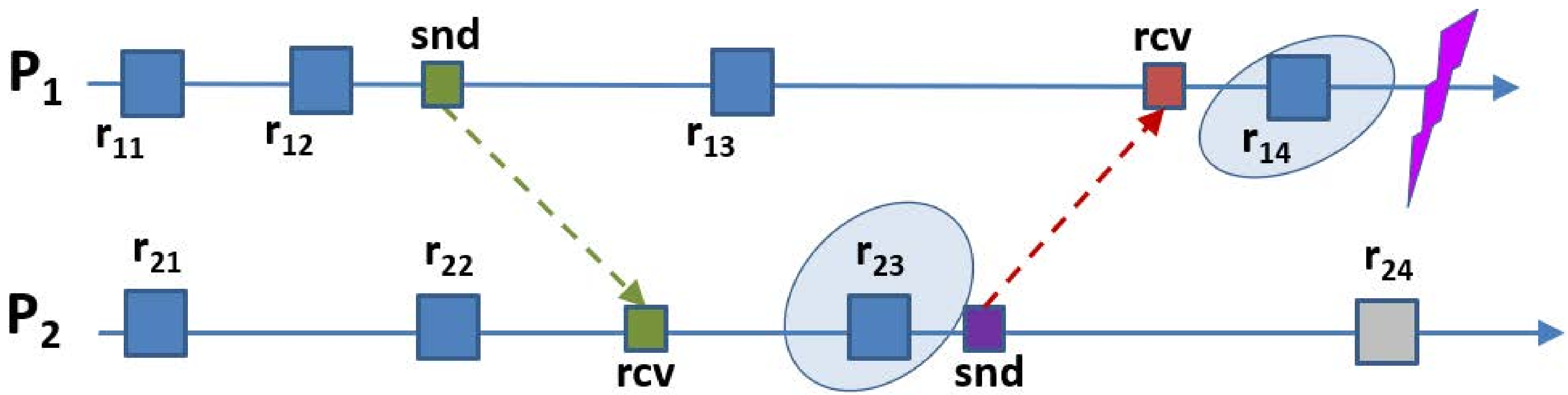
$\Longrightarrow$ Useful Procedure that keeps lot of information? Not always!

c.f. pg. VI-57

Problems using local Recovery Points only – 2



2. Crash in P_2: (r_{13}, r_{22}) **?** $\Longrightarrow$ **lost message** (will block P_2)
   Reset P_1 to r_{12} $\Longrightarrow$ roll-back to state (r_{12}, r_{22})



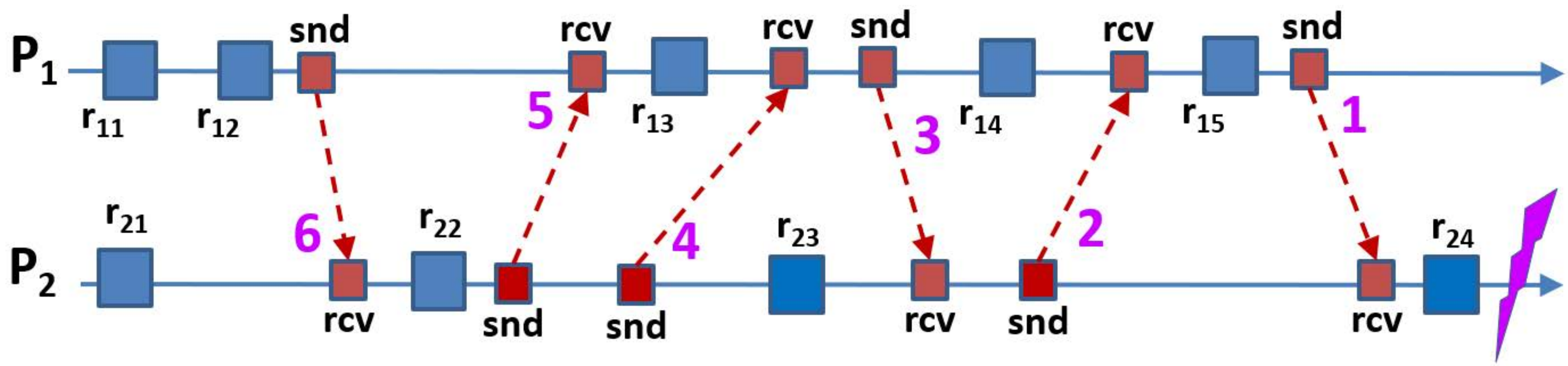
3. Crash in P_1: (r_{14}, r_{23}) **?** $\Longrightarrow$ **orphan message** (repeated `snd`)
   Reset P_1 to r_{13} $\Longrightarrow$ roll-back to state (r_{13}, r_{23})

Uncoordinated Procedure leads to Domino Effect

$\forall\ (P_i, P_j)$ check whether $\approx |\mathtt{SND}_{ij}| = |\mathtt{RCV}_{ji}|$?

- **Lost Messages** $|\mathtt{SND}_{ij}| > |\mathtt{RCV}_{ji}| \implies$ Reset SND process
- **Orphan Messages** $|\mathtt{SND}_{ij}| < |\mathtt{RCV}_{ji}| \implies$ Reset RCV process

until global recovery-point with *consistent information* is reached.



- Crash in $P_2 \implies (r_{15}, r_{24})$ includes Orphan 1 $\implies$ reset P_2
- (r_{15}, r_{23}) includes Orphan 2 $\implies$ reset P_1
- (r_{14}, r_{23}) includes lost message 3 $\implies$ reset P_1
- (r_{13}, r_{23}) includes lost message 4 $\implies$ reset P_2
- (r_{13}, r_{22}) includes Orphan 5 $\implies$ reset P_1
- (r_{12}, r_{22}) includes Orphan 6 $\implies$ reset $P_2 \implies (r_{12}, r_{21})$

Domino-Effect (worst-case)

Coordinated Procedure – General Considerations

$\Longrightarrow$ **Detect and avoid inconsistencies when recording**

1. **Lost Messages:** Record contents of **message channels**
   - fault-tolerance in network protocols and OS network interface
     e.g., store-and-forward until successful acknowledgement
   - repeating a SND is no big deal!

   $\Longrightarrow$ **Do not roll-back if** $|\mathtt{SND}_{ij}| > |\mathtt{RCV}_{ji}|$

2. **Orphan Messages:** repeated SND may disrupt receiver process
   avoid this kind of *inconsistency*

   $\Longrightarrow$ **Do rollback if** $|\mathtt{SND}_{ij}| < |\mathtt{RCV}_{ji}|$

Coordinated Approach:

- Coordinate local state recordings in order to avoid *domino effect*
- Provide **secure message channels**
  - System **active:** $\forall\ (P_i, C_{ij}, P_j)$ holds $|SND_{P_i}| = |C_{ij}| + |RCV_{P_j}|$
  - System **inactive:** $\forall\ (P_i, C_{ij}, P_j)$ holds $|SND_{P_i}| = |RCV_{P_j}|$

empty channels

System Model for Snapshot Algorithm – 1

- Process system $PS = \{P_1, \ldots P_n\}$
- **Complete Connectivity:** among each pair (P_i, P_j) of processes there exists a message channel C_{ij}
  1. Messages are **not** lost: middle-ware layer for lossless messaging
  2. Message channels have FIFO property, i.e., all channels (P_i, P_j) support proper message ordering.
- All processes and channels are *observable* (in theory)

Notation:

- $\forall P_i \in PS$ let LP_i be the local state of P_i
- **Observed Actions:** Send/Receive/Record state
  - $\texttt{snd}(m_{ij})$ sending the message m_{ij} in P_i
  - $\texttt{rcv}(m_{ij})$ receiving the message m_{ij} in P_j
  - LP_i recording the local state LP_i in P_i
- $\texttt{time}_i(event) \approx$ local *time* in P_i when $event$ occurs

System Model for Snapshot Algorithm – 2

Basic Principles:

1. **Which snd/rcv events are part of recording a local state:**
   (a) $\texttt{snd}(m_{ij}) \in LP_i \;:\Longleftrightarrow\; \texttt{time}_i(\texttt{snd}(m_{ij})) < \texttt{time}_i(LP_i)$
   (b) $\texttt{rcv}(m_{ij}) \in LP_j \;:\Longleftrightarrow\; \texttt{time}_j(\texttt{rcv}(m_{ij})) < \texttt{time}_j(LP_j)$

   $\Longrightarrow$ local state recording respects **causality** due to sequential execution order in P_i.

2. **Compare local states** of different processes P_i and P_j $(i \neq j)$:
   (a) **TRANSIT(**LP_i,LP_j**)** $:=$
   $$\{\; m_{ij} \mid \texttt{snd}(m_{ij}) \in LP_i \;\wedge\; \texttt{rcv}(m_{ij}) \notin LP_j \;\}$$
   msgs sent but not received w.r.t. compared states of P_i and P_j
   (b) **INCONSISTENT(**LP_i,LP_j**)** $:=$
   $$\{\; m_{ij} \mid \texttt{snd}(m_{ij}) \notin LP_i \;\wedge\; \texttt{rcv}(m_{ij}) \in LP_j \;\}$$
   msgs received but not sent w.r.t compared states, i.e. **Orphans**

System Model for Snapshot Algorithm – 3

Characteristics of global states:

1. **Global State** $GS := \{\ LP_1, LP_2, \ldots, LP_n\ \}$ such that holds:
$$\forall\ P_i \in PS\ \exists \text{ exactly one } LP_i \in GS$$
isolated recording of one local state for each process in PS

2. GS is **consistent** $:\Longleftrightarrow GS$ contains **no Orphans**, i.e.,
$\forall\ i \in [1:n]\ \forall\ j \in [1:n]$ holds: INCONSISTENT$(LP_i, LP_j) = \emptyset$

3. GS is **strong consistent** $:\Longleftrightarrow GS$ consistent **and**
$\forall\ i \in [1:n]\ \forall\ j \in [1:n]$ holds: TRANSIT$(LP_i, LP_j) = \emptyset$
combined local states hold all information due to empty channels

Note: If GS is consistent **and** there is an upper bound for all msg transfer times t_{max}:

`Record` GS`; Update` GS `after waiting` t_{max}

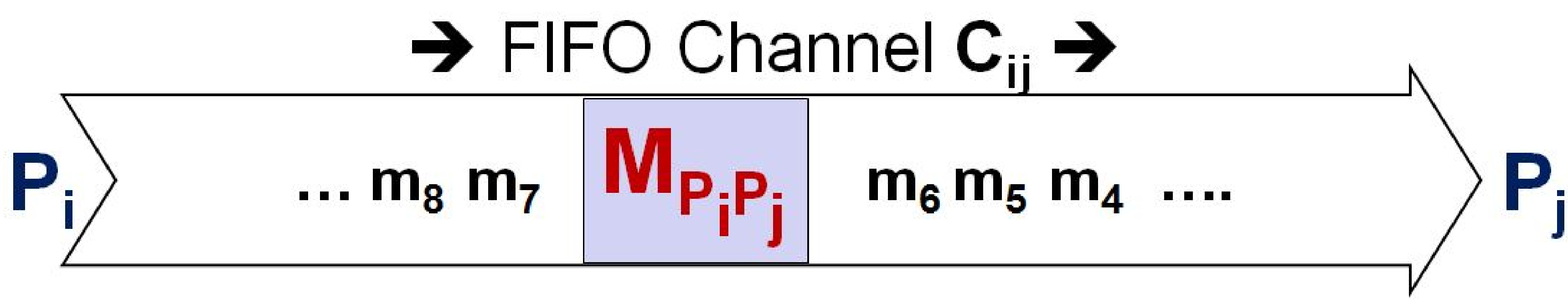
$\Longrightarrow$ strong consistent GS' is achieved easily.

Chandy-Lamport Snapshot Algorithm – 1

ACM ToCS 1985(1)

- **Precondition:** lossless FIFO channels
- **Idea:** Isolate `Phase`^{k}`; Record states; Phase`$^{k+1}$

1. Initiate state recording **locally** in any process P_i spontaneously
2. **Propagate** global recording via **marker** msg on all channels
   FIFO eases isolation between different phases by distinguishing among messages sent **before**/**after** local state recording.
   $\Longrightarrow$ Processes are allowed to proceed including `snd`/`rcv` actions.



3. Each P_i stores all msgs received after most recent recording LP_i
4. At the end of state recording $\longrightarrow$ send state to Coordinator

Chandy-Lamport Snapshot Algorithm - 2

- **Start Snapshot Algorithm:** in an arbitrary process P_i

  `record(`LP_i`);` (stores LP_i locally)

  `FORALL` $C_{i,j}$ `DO snd(`$M_{i,j}$`) OD;` (propagate execution)

- **Reaction** on a `rcv(`$M_{i,j}$`)` in process P_j (react at all times)

  `IF (NOT recorded(`LP_j`)) THEN`

  `STATE(`$C_{i,j}$`) := empty; record(`LP_j`); (store locally)`

  `FORALL` $C_{j,k}$ `DO snd(`$M_{j,k}$`) OD;` `(propagate)`

  `ELSE` `(update local state by all` $msg_{i,j}$ `received after local recording)`

  `STATE(`$C_{i,j}$`) := STATE(`$C_{i,j}$`)` $\cup$

  $\{msg_{i,j} \mid time(LP_j) < time(\texttt{rcv}(msg_{i,j})) < time(\texttt{rcv}(M_{i,j}))\ \}$;

  `FI;`

- **Combine to global state:** all processes send LP_i to coordinator after all marker have arrived in P_i (counter for $|\{C_{i,j}\}|$)

Observations – Algorithm Properties

▷ **Result:** consistent global state
no: strong consistent state, **but:** states of channels are known

◁ **complete global state** $\Longrightarrow \exists\, P_s$ connected to all processes in PS

▶ **Process Structure:** Initiation

important w.r.t. termination

- **Diffusion** phase $\approx$ when first marker arrives in a process P_j
  forward marker $M_{j,k}$ to all processes P_k directly connected to P_j
- **Contraction** phase $\approx$ update local state for 'late' messages
  no more forwarding of markers
- **Collection** phase $\approx$ send local state(s) to coordinator process

▶ **Distinguishing 'runs'** for different 'spontaneous' initiations:

1. Extend marker $M_{j,k}$ by initiator process number P_i: $M_{i,j,k}$
   P_i acts as coordinator $\Longrightarrow$ decentralized, independent 'runs'
2. Additional counter in P_i discriminates different runs by P_i

Properties of Snapshot Algorithm – 1

1. **Correctness:** Resulting state is consistent $\Longrightarrow$ **no Orphans**
   Proof: (indirect)
   Assumption: global state holds (at least) one Orphan message
   $\Longrightarrow \exists\ msg_{i,j}$ s.t. $\texttt{snd}(msg_{i,j}) \notin LP_i \wedge \texttt{rcv}(msg_{i,j}) \in LP_j$
   $\Longrightarrow P_i$ sends msg **after** recording local state LP_i and
   P_j has received msg **before** recording local state LP_j
   $\Longrightarrow$ **Contradiction:** Processes do not respect protocol:
   (a) $msg_{i,j}$ received **before** marker $M_{i,j}$ in $P_j \Longrightarrow$
      i. execution order: $\texttt{snd}(msg_{i,j})$; $\texttt{snd}(M_{i,j})$ in P_i **or** (illegal)
      ii. Channel $C_{i,j}$ is not FIFO (precondition not met)
   (b) $msg_{i,j}$ received **after** marker $M_{i,j}$ in P_j
      execution order: $\texttt{rcv}(M_{i,j})$; $\texttt{rcv}(msg_{i,j})$; LP_j in P_j (illegal)

   $\Longrightarrow$ **No Orphans iff algorithm is executed as programmed.**

Properties of Snapshot Algorithm – 2

2. **Termination:** for a single, distinguishable run
   Precondition: isolated steps in P_j terminate; finite transfer times
   - Diffusion phase initiated by P_i
     at most $|PS|$*($|PS|$-1) markers (maximum connectivity) as each P_j store and propagates exactly once (`THEN`) c.f. pg. VI-63
   - Contraction phase ends after a maximum of ($|PS|$-1) received markers at each process (full connectivity)
   - Collection phase: exactly ($|PS|$-1) messages LP_j go to P_i.

3. **Costs:** local memory, compute time and extra messages
   (a) $|PS|$ local states LP_i and $|PS|$*($|PS|$-1) channels
   (b) $|PS|$*($|PS|$-1) markers and ($|PS|$-1) local states

   Option: Record $B \subset PS$ of 'important' processes only,
   e.g., $Server$ for data bases, long-running computations

Importance of Message Channels for Snapshots

- Initiator does **not** reach all processes $\Longrightarrow$ **incomplete** snapshots
  direct connection or connectivity via transitive hull of msg channels
- explicit control messages needed to avoid incomplete snapshots
  otherwise: input data and actual process run may not contact all processes $\Longrightarrow$ not all processes will get markers.
  Tradeoff: *Additional control messages vs. quality of result*
- **Channel properties** are essential:
  1. No FIFO property $\Longrightarrow$ simulate message ordering c.f. VI.2
  2. No *piggy-backing* of messages or extra messages permitted?

  Without (1) **and** (2) $\Longrightarrow$ Algorithm has to **freeze entire system** until snapshot is recorded. (*How?*) Taylor 1989

  at least: prevent processes from sending messages during recording!

Note: Determining $TRANSIT$ is easy for FIFO channels with known maximum transfer times, otherwise hard to get. c.f. pg. VI-60

Global States and Causality

c.f. chapter VI.1

- **Cut** of process system PS $\approx$ *exactly one event for each* $P_i \in PS$

  $\Longrightarrow$ Recording LP_i $\forall$ $P_i \in PS$ results in a *cut* of PS

  Cut is consistent $\Longleftrightarrow$ all events are pairwise **concurrent**

  i.e. are not ordered w.r.t. *happened-before* relation $\xrightarrow{\sqsubset}$

  $\Longrightarrow$ **consistent global state is also a consistent cut**

  Orphan: `rcv` without `snd` implies causal chain through channels

- **Alternative Formalization:** (*Petrinet/process theories*)

  c.f. pg. VI-69

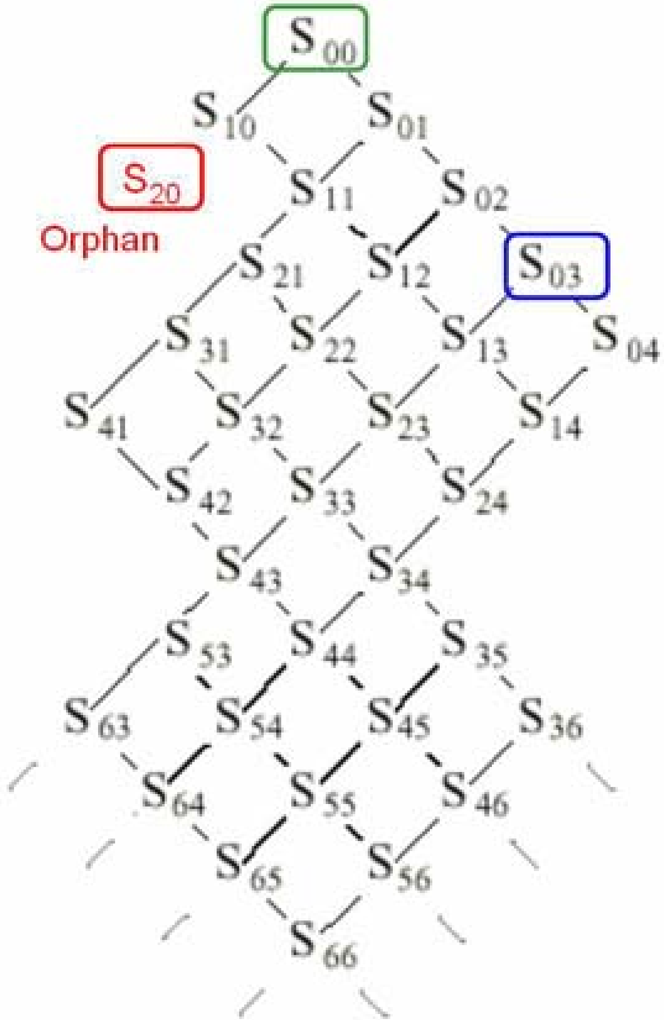
  1. $e_{ik} \in P_i$ subsumes its history$(e_{ik}) := e_{i1}e_{i2}\dots e_{ik}$, i.e., is identified with the prefix of the causal chain that leads to e_{ik}
  2. Cut$(PS) := \bigcup_{i\in[1:|PS|]}$ history(e_{ik_i}) (1 chain for each process)
  3. **Front** $:= \{e_{1k_1}, e_{2k_2}, \dots, e_{nk_n}\}$ most recent events of processes
  4. *Cut*(PS) *is consistent* $\Longleftrightarrow$

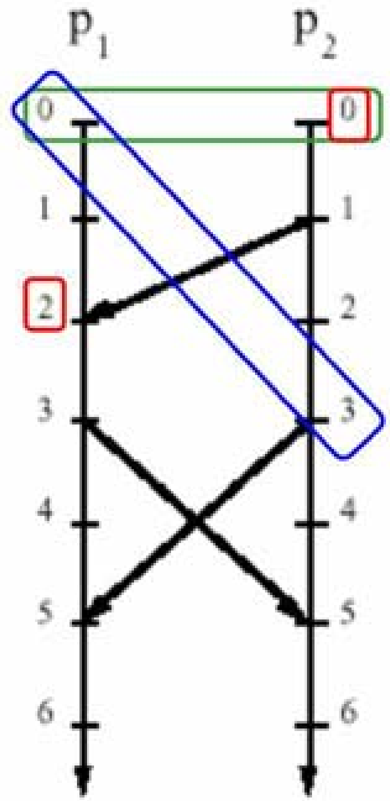
     $$(b \in \text{Cut}(PS) \wedge a \xrightarrow{\sqsubset} b) \Longrightarrow a \in \text{Cut}(PS)$$

  2. ensures local order in P_i; msgs are the 'critical' events

Example: Synchronic Distance among Processes

c.f. Colouris et al.: Distributed Systems Fig.10.15





- 'Lattice' of all permitted, combined states for P_1 and P_2
- more de-coupling, e.g., S_{25} not permitted w.r.t. consistency

End VI.4

VI.5 Detecting Global System States

Global Consistent View on a DS is always a challenge!

- **Non-volatile, permanent States**: Detection not time-critical
  - **System Termination** VI.5.1
  - **Deadlocks** VI.5.2
- **Volatile States**: How to obtain a consistent view? VI.4
  - local changes may occur spontaneously
  - detection algorithm may change the system behaviour
  - what about messages in $TRANSIT$?

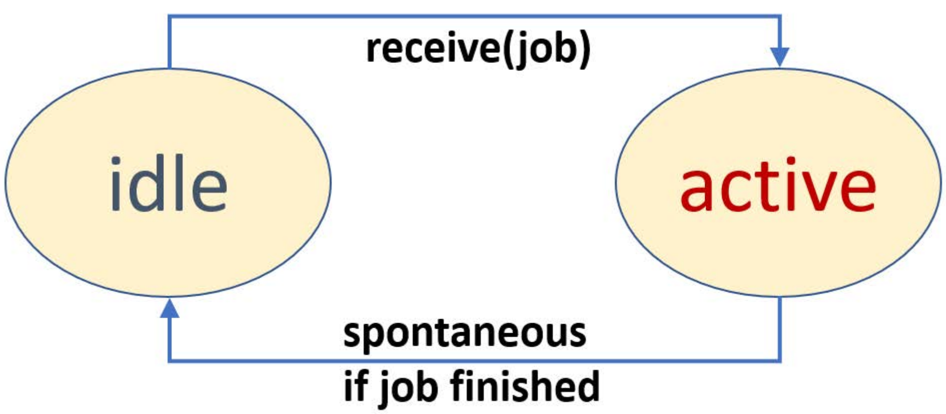
Additional Dimension: *State reachable in every run of the system?*

- Situation in the 'running' PS at hand $\Longrightarrow$ Debugging
- **Property**: for all PS when running a program $\Longrightarrow$ Proof
  - ∗ *Safety*–Properties: nothing bad will ever happen
  - ∗ *Liveness*-Properties: something wanted is eventually reachable

VI.5.1 Detecting System Termination

System Model: $PS = \{P_1, \dots P_n\}$

1. Message channel C_{ij} between each process pair (P_i, P_j)
   channels are 'robust', i.e. no messages are lost
2. Every $P_i \in PS$ is always in one of the two states { `idle`, `active` }:
   (a) `active` $\Longrightarrow$ Process may `snd`, `rcv` or `compute`
   (b) `idle` $\Longrightarrow$ `rcv` of messages only (**NO** `snd`!)
3. All State Changes permitted but:
   no spontaneous change to active based on internal reasons only



Why Termination? always a problem in de-centralized algorithms
$\Longrightarrow$ do not 'mix' termination detection with application

Basic Algorithmic Idea

- **Initial State:** Every process is `idle`, no messages
  - $\forall\ P_i \in PS$: $STATE(P_i) = \texttt{idle}$
  - $\forall\ P_i, P_j \in PS$: $TRANSIT(P_i, P_j) = \emptyset$ VI-60
- **Final State:** the same as initial state
- **Method:** unique **Monitor Agent** oversees all activities
  - $\exists$ **exactly one** $P_{mon} \in PS$ where P`.monitor()` $=$ `true`
  - **Weights** represent the level of work a process has to do
    1. $\forall\ P_i \in PS$ holds $\texttt{Weight}(P_i) \geq 0$
    2. **Algorithmic Invariant:** $$\sum_{P_i \in PS} \texttt{Weight}(P_i) = 1$$
  - P_{mon} starts with the initial job and the $\texttt{Weight}(P_{mon}) = 1$
  - Processes get part of the job along with part of the `Weight`
  - If a job is done, the corresponding weight is sent back to P_{mon}

  Result: $(\texttt{Weight}(P_{mon}) = 1) \implies$ System is terminated

Termination Detection Algorithm (Huang)

1989

```
STATE := idle; permitted_splits := S; W := 0;                               (0)
                    /* fixed number S avoids non-terminating distribution */
DO                     (non-deterministic choice among matching alternatives)
  rcv(P_from, WORK, Weight) ---> W := W+Weight;    /* may happen anytime */ (1)
     IF (STATE==idle) THEN STATE := active; FI;
  rcv(P_from, CTRL, Weight) ---> W := W+Weight;       /* Monitor P_mon only */  (2)
     IF (W==1) THEN snd(P_out,TERM); STATE := idle; FI;
  (STATE == active) --->   /* at least once, at most S times work distribution */ (3)
     IF (permitted_splits > 0) THEN permitted_splits := permitted_splits-1;
       < W_1, W_2 > := split(W);   /* where W_1, W_2 > MIN ∧ W_1 + W_2 = W */
       W := W_1; P := choose(PS \ {self}); snd(P,WORK,W_2);       FI;
  (STATE == active) --->                                /* Do Real Work */  (4)
     perform(WORK);
     IF (NOT(monitor())) THEN snd(P_mon,CTRL,W); W := 0; STATE := idle; FI;(4a)
OD;
```

Assumption: initial msg $(P_{out}, Work, 1)$ is received by P_{mon} in (1)

Outline of Correctnes and Termination Proof

1. P_{out} initializes algorithm with weight 1 and activates P_{mon} in (1)
2. `split` is a loss-less division $\Longrightarrow$ no weights are lost
3. **Invariant:** $W_{mon} + W_{active} + W_{work} + W_{ctrl} = 1$ (after init)
   - W_{active} $\approx$ Sum of weights from all P_i processes except P_{mon}
   - W_{work}, W_{ctrl} $\approx$ Sum of weights of corresponding messages
4. $P_i \in PS \setminus \{P_{mon}\}$ `active` $\Longleftrightarrow W_{P_i} > 0$
   - initially: `W` $=$ 0 and STATE $=$ `idle`
   - (`idle` $\mapsto$ `active`): only in step (1) adds Weight $>$ 0
   - (`active` $\mapsto$ `idle`): only in step (4a) and `W := 0` afterwards
5. Algorithm terminates $\Longleftrightarrow P_{mon}$ sends `TERM` to P_{out}
   Invariant 3. where $W_{active} + W_{work} =$ 0; $W_{ctrl} = 0$ after `TERM`

Termination: `perform`/msg-transfer times **finite** (Assumption)

- Only finite number of `split` steps in (3) due to ($MIN > 0$)
- No infinite delegation cycles in (3;1) due to limit S

VI.5.2 Distributed Deadlock Detection

Resources: *'Everything that processes compete for'* see also REST

- devices scheduled/data provided by the operating system
  Transparency $\Longrightarrow$ local as well as remote devices, data, ...
- Synchronization: Access to critical sections VI.2.2
- Messages/RPCs: blocking wait for a message or reply

Resources also an issue in traditional OS:

- OS Usage Protocol: Request – Wait – Hold – Free
- Deadlock problem if scheduling is bad or too optimistic
  Prevention vs. Avoidance, Detect and Resolve, Ignore

[**A. Tanenbaum, 1995**]:

Deadlocks in distributed systems are similar to deadlocks in single-processor-systems, only worse.

Goal: Efficient and Fair Resource Management

▶ **Global View** about state of all resources **required**

- **centralized** is easy to control but an architectural **bottleneck**
  $\Longrightarrow$ all requests go through a central Server/Manager
- **de-centralized** $\Longrightarrow$ problems w.r.t. **consistency of view**
  e.g., resource states change during message transfer times

◀ **Problem for both models:**
messages (requests, . . ., states) take too long, get lost, re-ordered c.f. pg. VI-70
$\Longrightarrow$ **General Problem requires handling of volatile states!**

Resource Model: (simplified for our algorithms)

- **non-consumable**, i.e. re-usable resources with **exclusive access**
- **no** multiple equivalent instances of resources (type vs. instance) VI-78

System Models for Processes PS & Resources RS

System State assignment of internal/external resources to processes that are either internal or on remote nodes

1. **Resource-Allocation Graph** $(PS \cup RS, E_{req} \cup E_{assign})$
   - $E_{req} \subseteq PS \times RS := \{(P_i, R_j) \mid P_i \text{ waits for } R_j\}$ **(requests)**
   - $E_{assign} \subseteq RS \times PS := \{(R_j, P_i) \mid P_i \text{ holds } R_j\}$ **(assigned)**

   Detailed: where are resources and who waits for which resource

2. **Wait-For-Graph** (PS, E) where $E \subseteq PS \times PS$ and $(P_i, P_j) \in E \;:\Longleftrightarrow\; \exists$ Resource R s.t. P_i is waiting for R and P_j holds R

   Abstraction: hold $\approx$ unambiguously
   wait $\approx$ maybe more than one process waits?

Deadlock $D \subseteq PS \iff (D, E \cap D \times D)$ contains at least one cycle
Note: Cycle implies deadlock iff there is only one instance/resource
$\Longrightarrow$ *Deadlock Detection is done via Cycle Detection*

Problem: RAG vs. WFG – Multiple Instances

Legend:

Circle $\in PS$

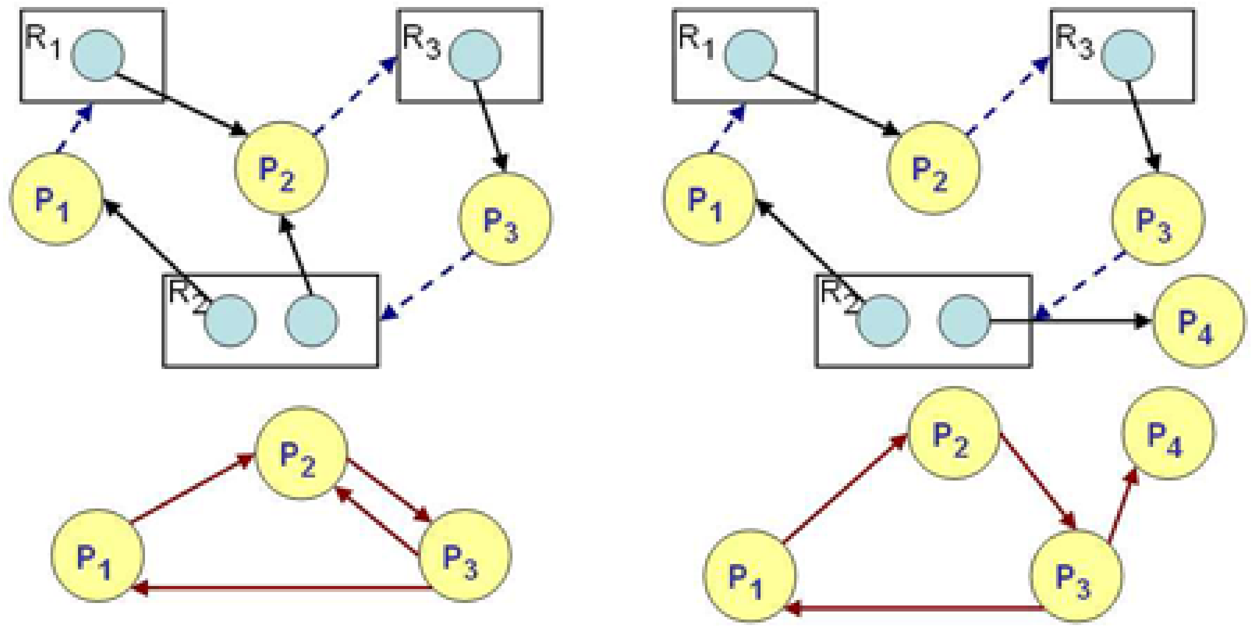
Square $\in RS$

Circle in Square

$\approx RS$-Instance

Edge (PS,RS) request

Edge (RS,PS) assign



RAG

vs.

WFG

WFG describes Wait-Relation, but no detailed cause w.r.t. resources

- ▶ more compact $\Longrightarrow$ easier to find cycles
- ◀ cycles may imply **false deadlocks**
  - ▷ **left:** conflict establishes cycle and a deadlock
  - ◁ **right:** conflict establishes cycle but *no deadlock* (P_4 may end)

Reason: Multiple resource instances imply **OR**–Relation for `wait`

Distributed Deadlock-Handling: Detect & Resolve

◀ **Centralized Solution:** RAG/WFG in a global control process

▶ **Adopted Centralized Solution:**

1. **Local** Resource-Allocation-Graph in each process VI-80
2. **Controller** combines local RAGs for detection; updates **either**:

(a) $P_i \in PS$ send always all changes to Controller push

(b) $P_i \in PS$ send local RAGs in fixed intervals push

(c) Controller requests local RAGs on demand pull

Problem: **false deadlocks** due to message timings $\Longrightarrow$

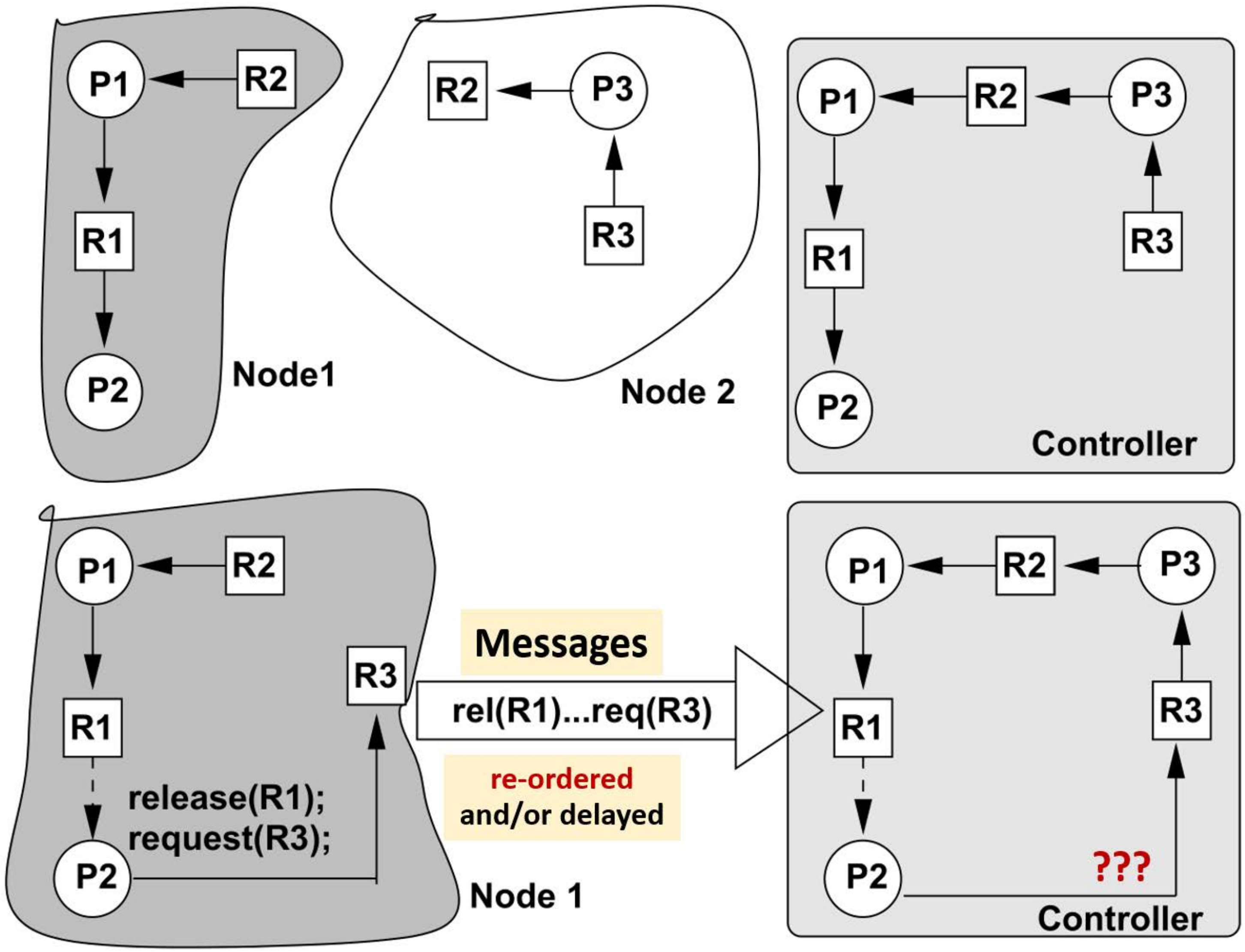
▷ ordered msg channels or request updated infos (timestamps)

▷ Controller holds 2 global RAGs and computes AND for all edges

$\Longrightarrow$ 2-Phase Ho-Ramamoorthy (1982)

▶ **Distributed Solution:** distributed construction of Wait-For-Graph VI-82
out of local WFGs and Edge Chasing Algorithm

$\Longrightarrow$ Chandy-Misra-Haas (1983)



False Deadlock identifiable: P_2 frees R_1; Termination Order $P_1 \longrightarrow P_3 \longrightarrow P_2$

Distributed Edge-Chasing–Algorithm

Chandy-Misra-Haas 1983

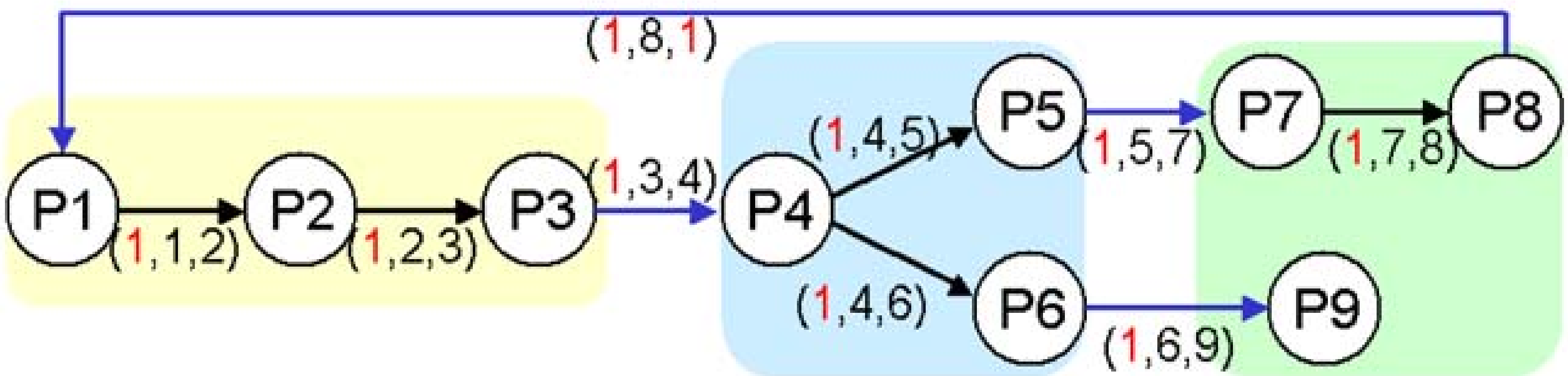
- Processes: request of multiple resources in a single request
  $\Longrightarrow$ Process may wait fo more than one other process
- Distinction: Waiting for **local** vs. **remote** resources
- No Controller $\Longrightarrow$ No global knowledge in a single process

Algorithmic Idea: How to detect cycles?
If process waits some defined time for a remote resource (blocked)
$\Longrightarrow$ Deadlock is 'suspected' $\Longrightarrow$ Msg exchange starts

- **Probe Msg:** $< i, j, k >$ $\approx$ $<$ Initiator, Sender, Destination $>$
- **Procedure:**
  - **Initiator** P_i sends Msg to all processes he **waits** for VI-82
  - also **waiting** destinations send msg to all those they wait for
  - If process P_i receives a message, it initiated $\Longrightarrow$ **Deadlock**
- **Optimization:** local `waits` are handled by local graph, not msgs
  msgs store route as a hint for deadlock resolution

Expl.: Edge-Chasing using three distributed Nodes

- local messages are avoided by looking into local Resource Graph
- msgs between nodes: explicit message-passing



Advantage: Construction of 'global knowledge' **On Demand**

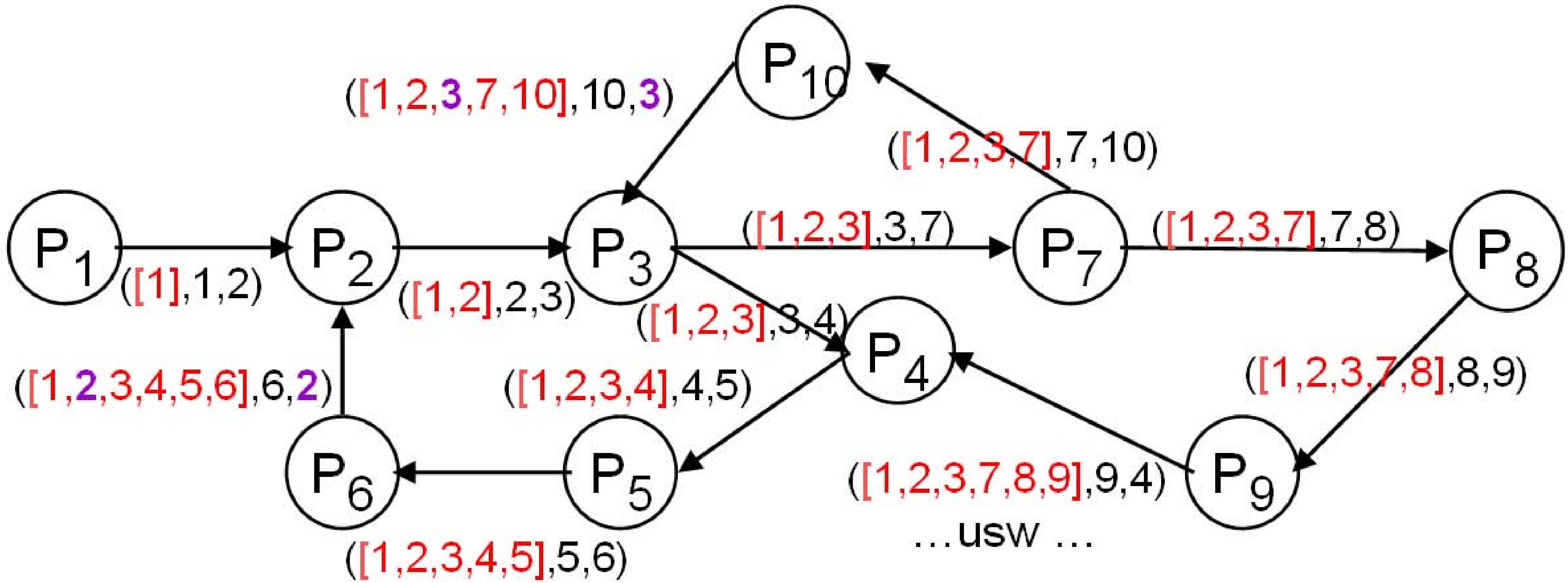
▶ Times for messaging not critical: processes are blocked anyway

◀ Waiting processes have to react to inquiry (watchdog) Thread

▶ single local controller with local RAG per node

Remark: *lots of different algorithms in literature* Shingal et al.

Expl.: Optimized Edge-Chasing stores Routes



Method: Propagate list of all nodes visited
Each node check whether it is the list

- ▶ Deadlocks are found independent of initiating process
- ◀ If multiple deadlocks are found $\Longrightarrow$ handling more complicated

Resolution: Preempt & Withdraw Resources

◀ **Problem:** preempted process loses work already done
recovery-points $\neq$ process start $\Longrightarrow$ costly VI.4
Acceptable iff recovery mechanisms are required anyway

▶ **Distributed Databases**: Transaction Management and Logging

- Precondition: unique time stamps used for prioritization
  $t_{P_1} < t_{P_2} \Longrightarrow P_1$ older than $P_2 \Longrightarrow P_1$ loses more work
  $\Longrightarrow$ **Older Process will not be terminated!**
- **How to avoid cyclic 'killings'?**
  ◁ **Wait-Die**: wait only for **higher** time stamps
  $P_{old} \longrightarrow P_{young} \approx \texttt{wait} \mid P_{young} \longrightarrow P_{old} \approx \texttt{kill}(P_{young})$
  Problem: $(\texttt{kill; restart})^+ \Longrightarrow$ thrashing effect
  ▷ **Wound-Wait**: wait only for **lesser** time stamps
  $P_{old} \longrightarrow P_{young} \approx \texttt{kill}(P_{young}) \mid P_{young} \longrightarrow P_{old} \approx \texttt{wait}$
  Advantage: `kill; restart; wait` much less overhead End VI.5

Remark: *Lots of literature in distributed OS and DBS.*

VI.6 Distributed Coordination

▶ Arbitrarily distributed application system with *distributed*

- *control* to ensure robustness without bottlenecks
- *data* due to replication-based transparency
- *compute load* due to efficiency and feasibility reasons

◀ Basic level of agreement and consensus essential for, e.g.,

- mutual exclusion and consistency mechanisms for replicated data
- centralized/de-centralized organization of common decisions

Typical coordination problems in distributed systems

1. **Election**: Determine new unique coordinator after crashes
2. *Agreement* about common 'global' state in the context of errors
3. *Commitment* about executing collective actions, e.g. *transactions*

additi-
onal
MSc
topics

Note: Dynamic process systems/groups with ever changing process numbers are especially 'hard', e.g., mobile or P2P systems.

VI.6.1 Leader Election

Motivation: Leader Election is the basis for

- distributed algorithms that use some kind of centralized organization, e.g., mutual-exclusion, resource handling, deadlock or termination detection
- fault-tolerance in these algorithms by allowing for de-centralized replacement of crashed coordinators, servers etc.

$\Longrightarrow$ **Requirements for any Election Algorithm:**

1. may be initiated by **any** process in the system
2. may be initiated **in parallel** by different processes

Reason: a priori unknown when and where a process crashes

3. terminates always appointing **exactly one Leader**
   i.e, all processes know their state w.r.t. election: { `leader`, `lost` }

System Model for Leader Election

- **Lossless** message transfer
- Each $P_i \in PS = \{P_1, \ldots P_n\}$ knows about **all 'names'** in PS reali-stic ?
- All processes $P_i \in PS$ are **a priori equal** *

  $\Longrightarrow$ **Each process** is able to act as a coordinator
- **Priorities** are defined by arbitrary **total, strict order** $<$ on PS

  e.g., totally ordered process indices $P_1 < P_2 < \ldots < P_n$

  $\forall\ M \in \wp$(PS) is $P_i \in M$ assigned the highest priority $:\Longleftrightarrow$

  $$i = MAX(\{j \mid P_j \in M\})$$ *

At any time, PS is partitioned into two process sets:

1. $UP(PS)$ $\approx$ active processes
2. $DOWN(PS)$ $\approx$ crashed processes

State changes are **spontaneous and not coordinated**

Caution: Even change from *crashed* $\longrightarrow$ *active* is **not** predictable

Additional Challenges

- **Distinction:** crashed nodes vs. lost messages $\Longrightarrow$
  lossless communication with **maximum response time** t_{max} known
  (msg transfer plus time to react, e.g., via 'watchdog' thread)
- **Elections may be initialized in parallel**
  set of nodes waiting for a response governed by a timeout
  e.g., access to a critical section: missing `WAIT`/`OK` messages c.f. III-66
- **Formerly Leader is re-started** and becomes active again
  Who is assumed to stay/become coordinator in this case?

System structure and fault tolerance

1. **complete** reachability $\Longrightarrow$ *broadcast*
2. **partial** reachability $\Longrightarrow$ special protocols needed
   Example: Algorithms based on ring structures

Note: Partitioning due to channel crashes is most critical

Challenge: Determine a **single**, unique coordinator, s.t.
all $P_i \in UP(PS)$ **agree in**
the name of the (new) coordinator process

Typical Algorithms: $Bully$-Algorithm
Ring-Algorithm

c.f. pg. VI-93

Bully Algorithm (Garcia-Molina 1982)

- **Precondition:** Each active process is able to reach **all** others
- **Idea:** Prioritize the process with the highest process index
- **Method:** `timeout` in a process $\Longrightarrow$
  initiate re-election for all processes with higher indices
  stop re-elections from processes with lower indices
  single process that has highest index $\Longrightarrow$ is elected

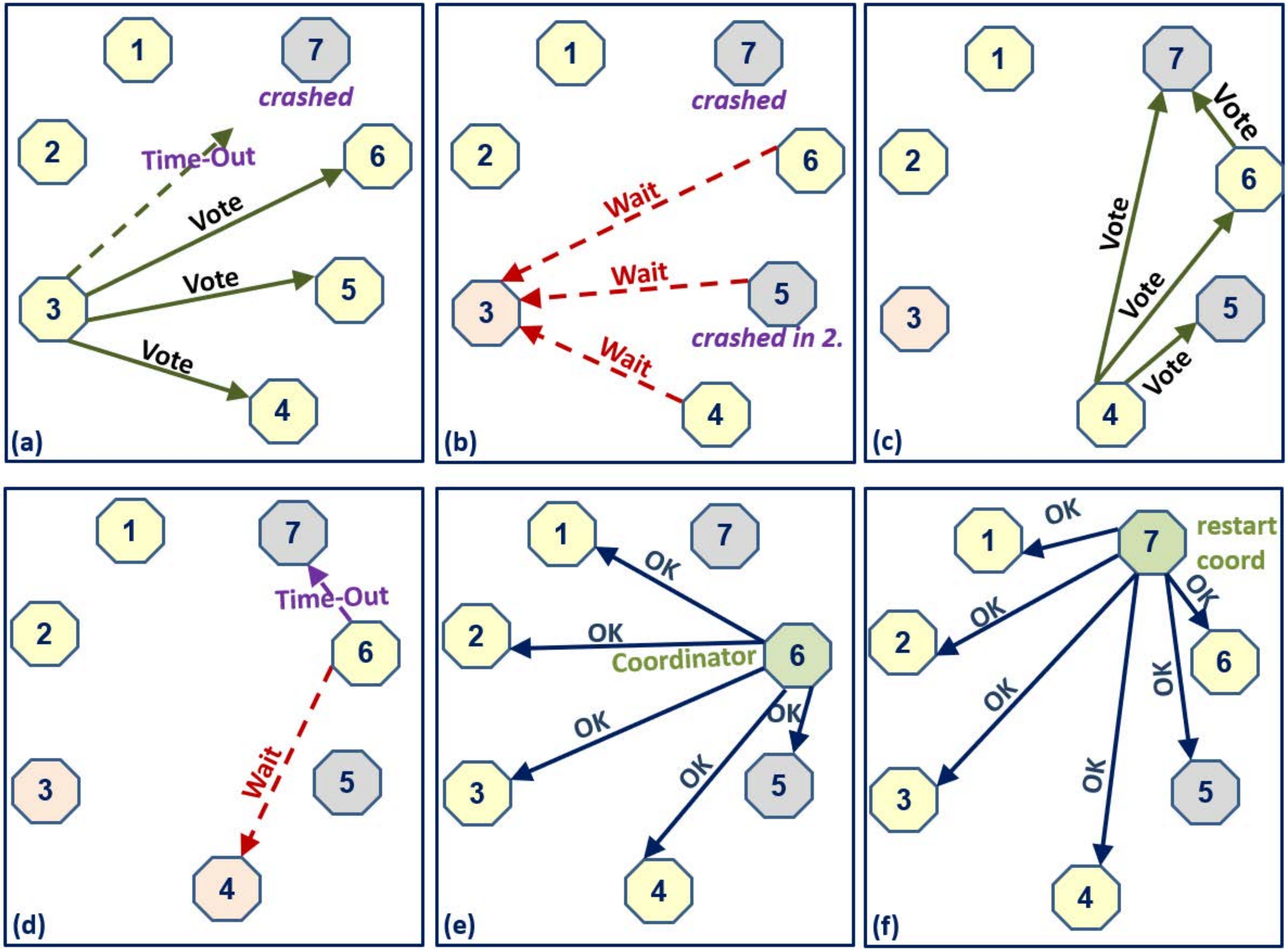
Note: start of multiple re-elections in parallel possible
after re-start, the 'former' coordinator is favored to become
the new coordinator again ('bullies' the current coordinator)

Each active process P_i reacts to the following 6 types of events:

1. **Time-out** of a msg to (last known) coordinator $\longrightarrow$ GOTO 2.
2. **Initiate re-election:**
   FORALL $\{P_j \in PS \mid j > i\}$ DO snd(P_j,VOTE) OD; /* higher up? */
   ELECTION := true; RESPONSE := $\emptyset$; Coord := undef;
   Wait(T) for EVENTS of $\{$ 3, 5 $\}$; (timeout T)
   IF (RESPONSE == $\emptyset$) /* all higher procs down */
   THEN FORALL $\{P_j \in PS \mid j < i\}$ DO snd(P_j,OK) OD; /* self */
   ELSE Wait(T') for EVENTS OF $\{$ 4, 5 $\}$; /* higher elect */
   IF (Coord == undef) THEN 2. FI
   ELECTION := false;
3. rcv(P_j,WAIT) $\longrightarrow$ RESPONSE := RESPONSE $\cup$ $\{P_j\}$;
4. rcv(P_j,OK) $\longrightarrow$ Coord := P_j; /* new coordinator? */
5. rcv(P_j,VOTE) $\longrightarrow$ snd(P_j,WAIT); /* stop 'lower' election */
   IF NOT(ELECTION) THEN 2. FI; /* vote */
6. **Recover** (from crash) $\longrightarrow$ GOTO 2.

Note: Process crashes after WAIT $\Longrightarrow$ never 4.: re-start election

Example: Bully Algorithm



Assessment of Bully Algorithm

- Nomen est Omen; very simple structure
- works despite multiple node crashes during election
- parallel elections cause no problems and will be stopped 'early'
  $\Longrightarrow$ processes with higher process indices determine overall result
- a single run terminates always due to finite number of messages

◀ Lots of crashes $\Longrightarrow$ triggers incessantly re-elections!
  $\Longrightarrow$ Determination of **timeouts** is critical for success

◀ **Costs** $\mathcal{O}(|PS|^2)$ messages ($worst\text{-}case$):
  UP $= PS$; P_n crashes;
  $P_1, \ldots, P_{n-1}$ observe crash in parallel and start $(n-1)$ elections;
  P_n is restarted again
  $\sum_0^{n-1}$ `VOTE` $+ \sum_0^{n-1}$ `WAIT` $+ (n-1)$ `OK` messages

Conclusion: rather costly w.r.t. number of messages exchanged

Ring Algorithm

1. **Naive** version using direct neighbors in a ring
2. **Efficient** version collecting information in a logical tree structure

Ring-Algorithm (LeLann 1977)

- **Precondition: uni**-directional ring for communication among active processes $P_1 \mapsto P_2 \ldots \mapsto P_n \mapsto P_1$

  $P_i.next()$ determines current next **active** neighbor $\in UP$

  $\Longrightarrow$ *strong precondition used here!*
- **Idea:** Use maximum index in **list of processes** w.r.t. ring direction
- **Method:** timeout in process $P \Longrightarrow$

  initiate re-election; start with singleton list $[P]$

  handover and extend list

  after complete circle $\Longrightarrow$ propagate coordinator list

Note: start of multiple re-elections in parallel possible

restart of a former coordinator initiates new election

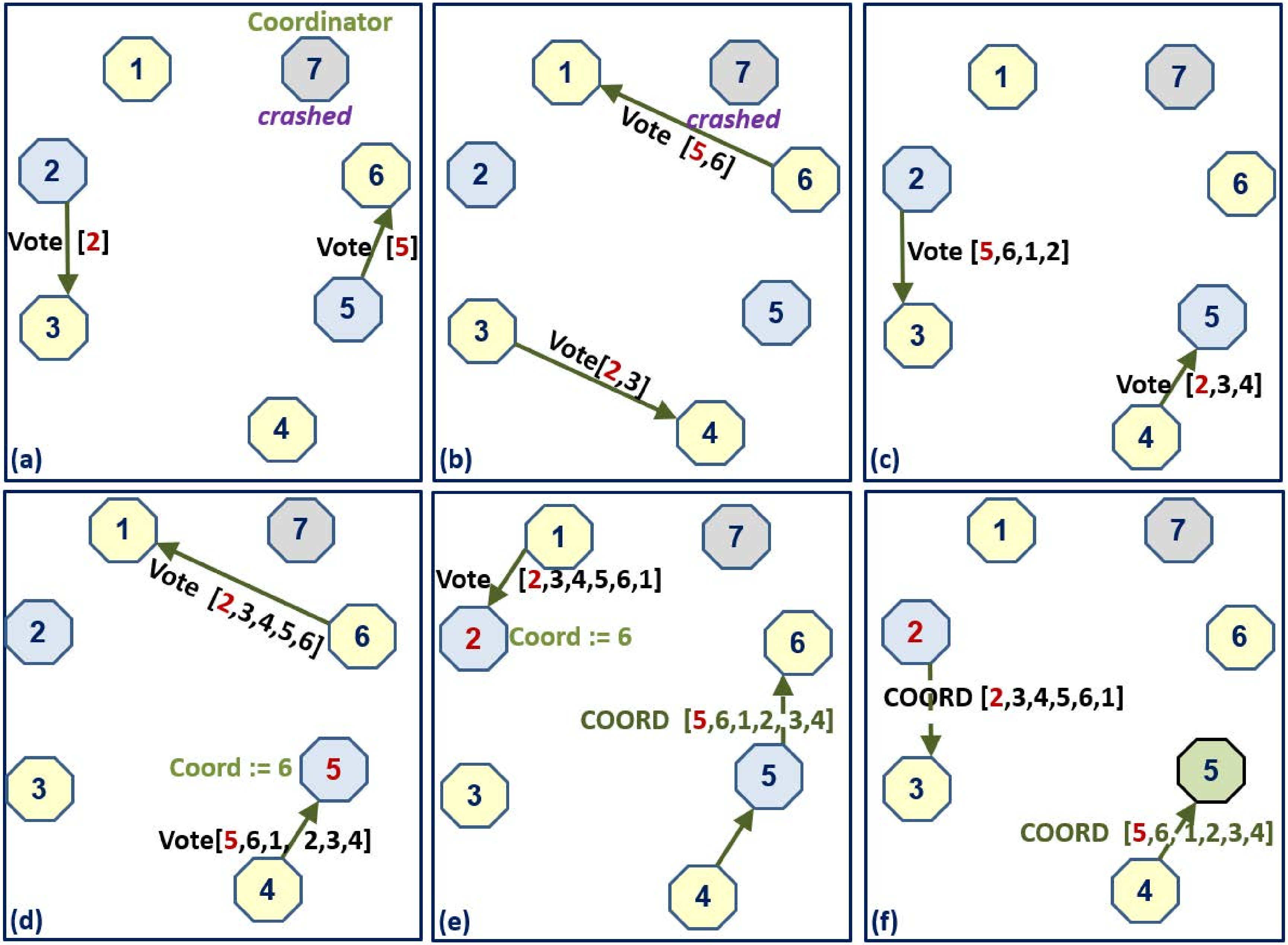
LeLann Ring Algorithm

Each active process P_i reacts to the following events:

1. **Time-out** of a msg to (last known) coordinator $\longrightarrow$ GOTO 2.
2. **Initiate Election:** `snd(`$P_i.next()$`,VOTE`$[i]$`);`
3. `rcv(`P_j`,VOTE`$[List]$`)` $\longrightarrow$ `/* transform */`
   `IF` $i \in List$ `THEN snd(`$P_i.next()$`,COORD`$[List]$`);`
   `ELSE snd(`$P_i.next()$`,VOTE`$[i : List]$`);`
   `FI;` /* extend */
4. `rcv(`P_j`,COORD`$[List]$`)` $\longrightarrow$ `/* term/propagate */`
   `Coord :=` $MAX([List])$
   `IF` $i \neq List.front()$ `THEN snd(`$P_i.next()$`,COORD`$[List]$`);`
5. **recover** after crash $\longrightarrow$ GOTO 2. /* no privileges */

Note: $VOTE$ $\approx$ ask around; $COORD$ $\approx$ propagate result
Termination: initiating process aborts further propagation

Example: LeLann Ring Algorithm



Assessment of LeLann Algorithm

▶ simple structure

▶ parallel elections cause no problems and achieve common result MAX

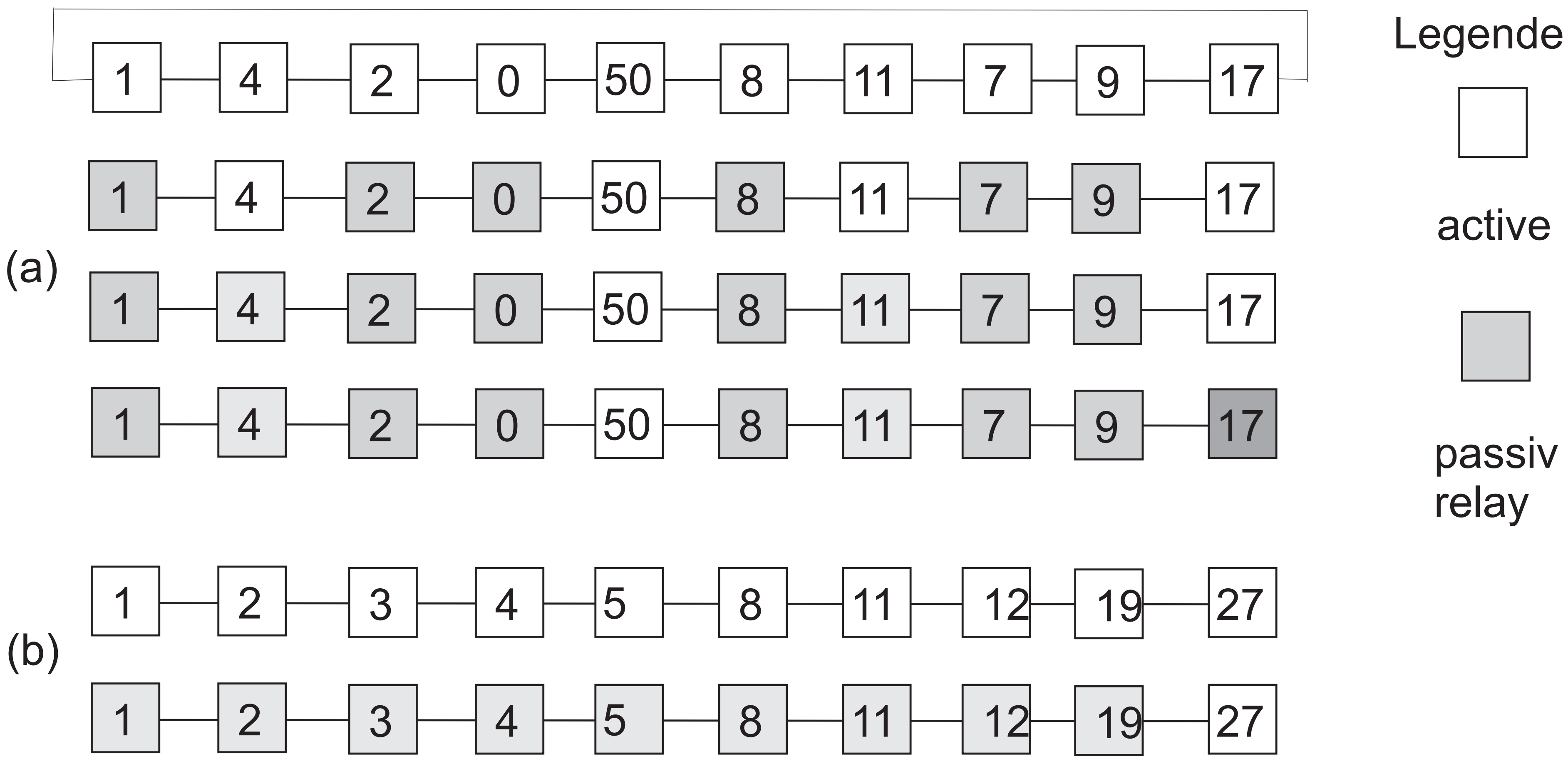
◀ **Costs:** each election requires 2*($|PS|$-1) messages
worst-case ($|PS|$-1) parallel election procedures
($|PS|$-1) * (2*$|PS|$-1)) messages $\Longrightarrow \mathcal{O}(|PS|^2)$

Conclusion: rather costly and requires optimizations

Worst-case Optimization: (Peterson 1982) ACM ToPLaS, (10) 1982

- tree-like reduction of ring structure for parallel elections
- active $\approx$ participate in election/ relay $\approx$ transfer messages only
- each step cuts number of active processes in half (at least) via
  - **bi**-directional ring: MAX($\{P_{i\ominus 1}, P_i, P_{i\oplus 1}\}$) (left/right) c.f. pg. VI-97
  - **uni**-directional ring: MAX($\{P_{i\ominus 2}, P_{i\ominus 1}, P_i\}$) (2 predecessors)

Example: Peterson Ring-Election Algorithm



End of VI.5

VI.6.2 Agreement Protocols – Outlook

Simplified Process Model

- PS with n Processes (nodes)
- **robust, loss-less direct communication** $\forall\ (P_i, P_j) \in PS^2$

$\Longrightarrow$ no special handling based on node where error occurs

Reason: Network Partitioning in central node, e.g., star network
Error in name server vs. user node
No subsequent errors due to transitive message routing

ERROR (Fault): *Deviation of expected system behavior* in

◀ **General** systems, i.e., synchronous as well as asynchronous: no or unexpected reaction of nodes or msg channels VI-100

◀ **synchronous** systems additionally via **timing** errors, e.g., clock drift exceeds tolerable limit; msg transfer times time for computing or replies exceeds limit

c.f. I-33

Why is Agreement important in DS?

◀ **Quality of Service** assumptions are not met

Expl.: E-commerce, online orders . . .

guaranteed delivery time exceeded; order not processed

◀ **Distributed Algorithms fail** due to preconditions not met

$\Longrightarrow$ distributed infrastructure no longer robust

Expl.: Algorithms waiting for msgs will block

Bully–Algorithm using faked process indices

$\Longrightarrow$ Fault tolerance is important

- **Replication** of active and passive components chap. VII
- Saving states as recovery points; Logging (transactions) VI.4
- **Reduce impact of faulty components**
  - ∗ Determine which parts are faulty and which are ok
  - ∗ Secure functionality of system parts working correctly

Errors and Faults: Causes and Classes

- **Connections:** lost, duplicated, corrupted msgs
- **Nodes:** different levels of impact for errors, esp.
  1. **Fail stop**: P_i **identifiable** permanent down
  2. **Crash fault**: P_i **not identifiable** permanent down
     $\Longrightarrow$ Process does not send or reply in the future
  3. **Send Omission fault**: not all msgs (to all destinations) are sent broad-cast
  4. **Receive Omission fault**: not all msgs arriving are accepted
     Impact: content loss plus blocking in synchronous interaction
  5. **Byzantine fault**: Unpredictable behavior of nodes **!!** worst case
     sometimes correct, sometimes faulty behavior
     omits/**manipulates** some msgs, even generates unexpected msgs

AGREEMENT: Make sure that **all non-faulty nodes** get always the correct information

$\Longrightarrow$ *Prohibit or Reduce the effects of faulty nodes.*

Agreement Levels for Byzantine Faults

1. **Byzantine Agreement** (**single**-source broadcast)
   A **single** P_i propagates a fixed value v_i and **all** non-faulty processes agree on a **single value** V (If P_i is not faulty $\Longrightarrow V = v_i$)
2. **General Consensus** (**multiple**-source broadcast)
   Each P_i propagates **it's own** fixed value v_i (values may differ) and **all** non-faulty processes agree on a **single value** V arbitrary ?
   (If all v_i are the same in all non-faulty processes $\Longrightarrow V{=}v_i$)
3. **Interactive Consistency** (**multiple**-source broadcast)
   Each P_i propagates **it's own** fixed value v_i (values may differ) and **all** non-faulty processes agree on a **Vector** $(V_1, V_2, \ldots, V_n)$ of values (If P_i is not faulty $\Longrightarrow$ entry V_i of the vector V is v_i) see Colouris 11.5

Note: Implementing 1. for all processes implies 3.
Implementing 3. plus a global majority function implies 2.

Byzantine Generals – 'History?'

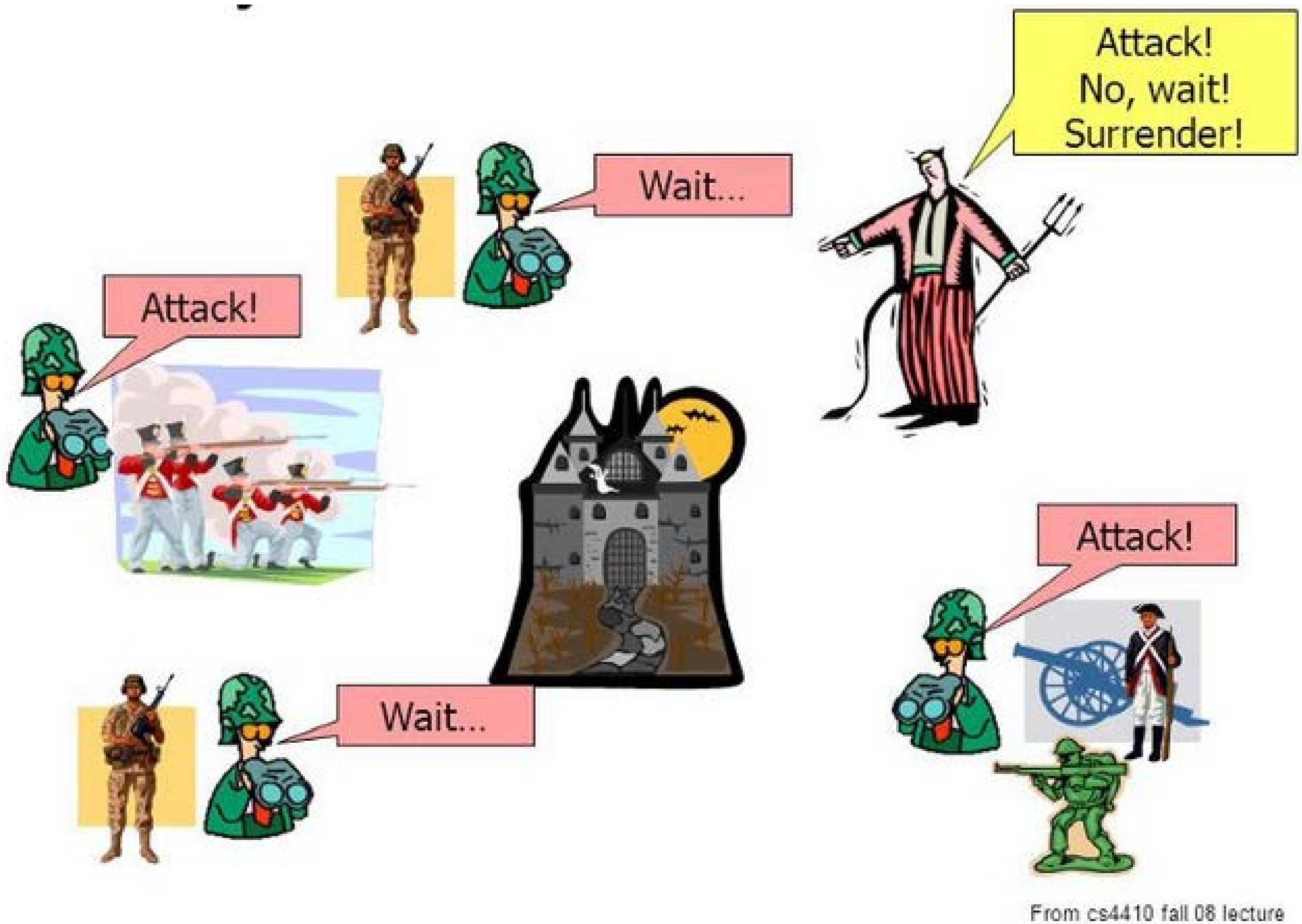


Fig.: `gauthamzz.github.io/tendermint.html#byzantine-fault-tolerant`

Byzantine Agreement - Basic Situation

- ▶ A single P_i wants to establish a consensus about value v_i in PS
- ▶ Arbitrary node failures are 'expected'; no messages lost VI-100
- ▶ All processes use the same symmetrical algorithm
- ◀ Messages may be manipulated/faked
- ▷ **No** use of authorization or signatures VI-108

Algorithm: Exchange values to guarantee **forming a majority**

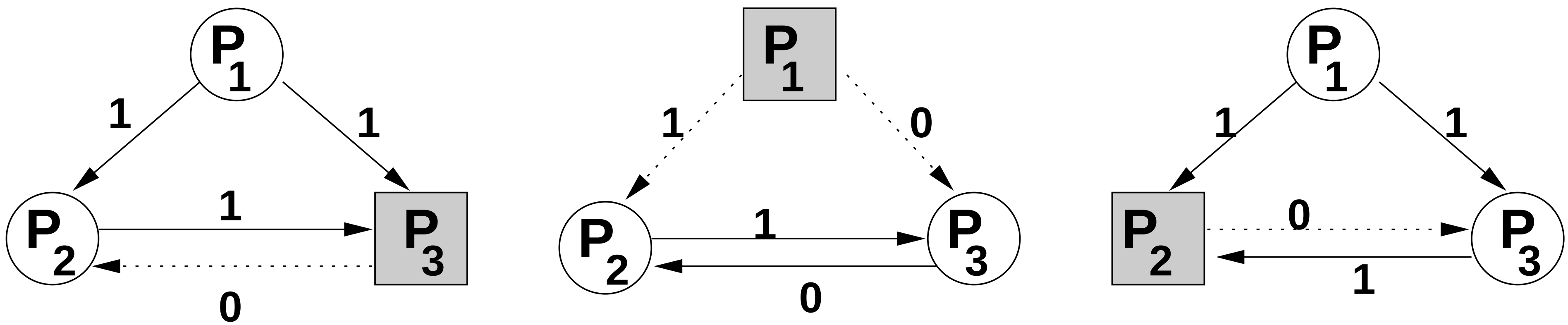
Problem:

- ◀ Nodes may manipulate msgs before forwarding
  - $\Longrightarrow$ multiple interaction (rounds) used to detect faked msgs
  - $\Longrightarrow$ message exchange is very costly
- ◀ **Only** $m \leq \frac{n-1}{3}$ $(ceiling)$ of **faulty processes** in n nodes are tolerable, i.e., $n = 3m + 1$ processes may compensate for m faults VI-104

  Example: 1 out of 4 is ok; 1 out of 3 not;
  2 out of 7 processes etc.

Example: 3 Processes with 1 Faulty Process

P_2: P_1, P_2 ok, P_3 fault $\Longrightarrow$ P_2 has to choose P_1 value left
P_2, P_3 ok, P_1 fault $\Longrightarrow$ P_2 has to choose P_1 value (same algorithm) center



P_3: P_1, P_3 ok, P_2 fault $\Longrightarrow$ P_3 has to choose value of P_1 right
P_2, P_3 ok, P_1 fault $\Longrightarrow$ P_2 has to choose value of P_1 (same alg.) center

$\Longrightarrow$ **No Agreement** among P_2 and P_3 if P_1 is faulty

Generalization: $(m > 1)$ reduces to 1-of-3 problem by *contradiction*
$\Longrightarrow$ general algorithm would work also for $(m = 1)$

Oral-Message-Algorithm: Lamport/Shostak/Pease

ACM ToPLaS 1982

Method: Recursive Exchange as Remote Procedure Calls

- **Arbitrary Call** `om(m,PROCS,p)`
  `m` $\approx$ Max-Faults; `PROCS` $\subseteq PS$ participate; `p` $\approx$ starts call

```
om(m,PROCS,p) ::= PROCS := PROCS \ { p };
    FORALL P ∈ PROCS DO snd(P,val_p) OD;                      (1)
    IF (m > 0) THEN
        FORALL P ∈ PROCS DO RPC(P,om(m-1,PROCS,P)) OD;        (2)
        FORALL P ∈ PROCS DO RPC(P,majority(P,PROCS)) OD;      (3)
    FI;
majority(P,PROCS) ::= val_p := Majority value from all P' ∈ PROCS;
```

- P $\in$ PS **reacts**: after `rcv(p,val`$_p$`)` start own RPC call
- **initial Call:** `om(m,PS,P`$_{init}$`)` of Initiator P$_{init}$ using `val`$_{P_{init}}$

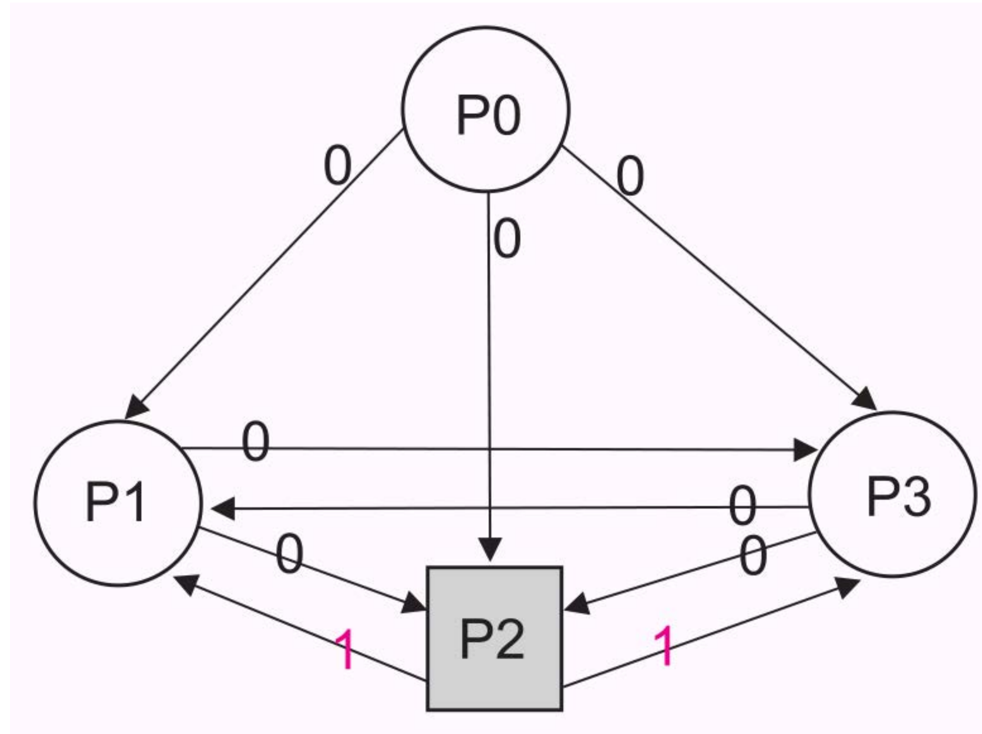
Costs: `om(m,PS,p)` $\approx$ |PS|-1 RPCs `om(m-1,`$PS\setminus\{p\}$`,P)` ...

Expl.: $m = 1$; 4 processes $PS = \{P_0, P_1, P_2, P_3\}$: `om(1,PS,`$P_0$`)`

1. Case: **Initiator P_0 is non-faulty**

   (a) Let `val`$_{P_0} = 0$
   P_0 sends 0 to $\{P_1, P_2, P_3\}$
   `val`$_{P_1} =$ `val`$_{P_3} = 0$

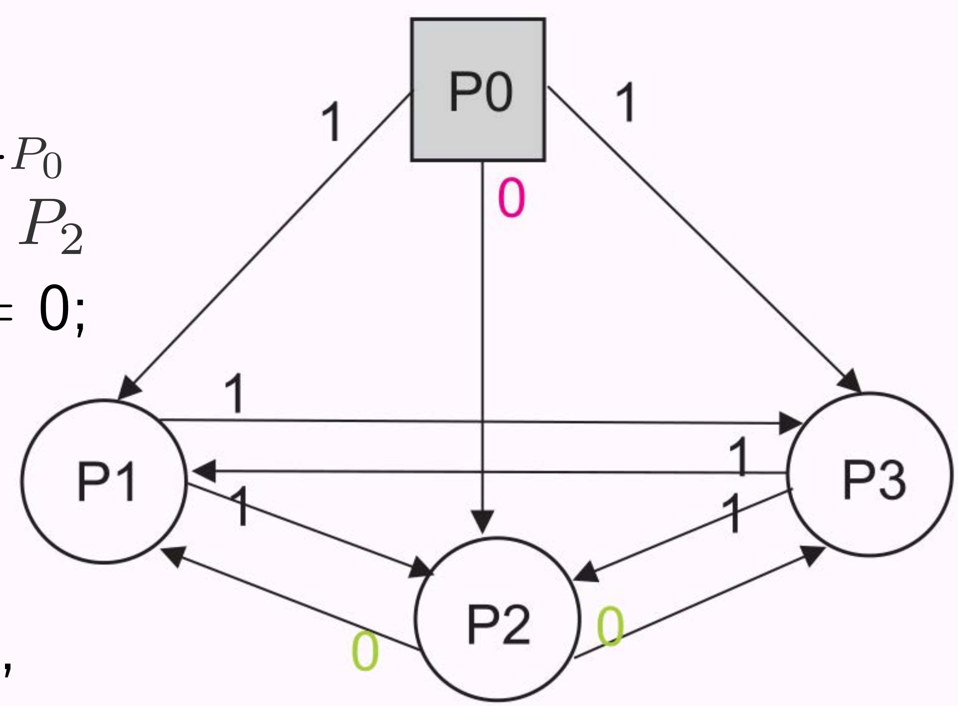
   (b) in P_1 `om(0,`$\{P_1, P_2, P_3\}$`,`P_1`)`
   in P_2 `om(0,`$\{P_1, P_2, P_3\}$`,`P_2`)`
   in P_3 `om(0,`$\{P_1, P_2, P_3\}$`,`P_3`)`
   forward correct in P_1 and P_3
   forward faked in P_2



   (c) `majority`
   in P_1 using $< 0, 1, 0 > \implies 0$ /* values from $< P_1, P_2, P_3 >$ */
   in P_2 using $< -, -, - >$??
   in P_3 using $< 0, 1, 0 > \implies 0$
   $\implies \{P_0, P_1, P_3\}$ agree on the **same value** 0

Expl.: $m = 1$; 4 processes $PS = \{P_0, P_1, P_2, P_3\}$: `om(1,PS,P`$_0$`)`

2. Case: **Initiator P_0 itself is faulty**



(a) arbitrary (unknown) value in $\texttt{val}_{P_0}$
P_0 sends 1 to $\{P_1, P_3\}$, but 0 to P_2
$\texttt{val}_{P_1} = \texttt{val}_{P_3} = 1$ but $\texttt{val}_{P_2} = 0$;

(b) in P_1 `om(0,`$\{P_1, P_2, P_3\}$`,`P_1`)`
in P_2 `om(0,`$\{P_1, P_2, P_3\}$`,`P_2`)`
in P_3 `om(0,`$\{P_1, P_2, P_3\}$`,`P_3`)`
forward correct in P_1, P_2 and P_3,
but different values $\texttt{val}_{P_i}$

(c) `majority`
in P_1 mit $< 1, 0, 1 > \implies 1$ /* values from $< P_1, P_2, P_3 >$ */
in P_2 mit $< 1, 0, 1 > \implies 1$
in P_3 mit $< 1, 0, 1 > \implies 1$
$\implies \{P_1, P_2, P_3\}$ agree on the **same value** 1

Algorithms: $3m+1$ Processes – m tolerable faults

- **Lamport/Shostak/Pease 1982** ACM ToPLaS VI-105
  $m+1$ Exchange rounds; overall $\mathcal{O}(|PS|^m)$ Msgs
- Dolev/Reischuk: $2m+1$ Rounds; $\mathcal{O}(|PS| * m + m^3)$ Msgs JACM 1985
- Gary/Moses (1993): $m+1$ rounds; number of msgs polynomial

$\Longrightarrow$ m tolerable faults require $m+1$ deterministic exchanges Fischer/Lynch 1982

$\Longrightarrow$ Trade-Off: Number of messages vs. Number of Rounds

$\Longrightarrow$ *Algorithms are too costly for most situations*

More Efficient Solution $\Longrightarrow$ **Advanced System Model**

- Use forgery-proof **Signatures** for all messages
- Protect message channels from eavesdropping and manipulation

$\Longrightarrow$ *Fakes when forwarding messages can be detected*

Expl.: Signatures allow even for 1-out-of-3 solution VI-104
Exchange $P_2 \longleftrightarrow P_3$ exposes inconsistent msgs from P_1 End VI.6.2

VI.6.3 Outlook: There are a lot more Algorithms

- Coordination in flat and nested transactions: 'Commit Protocols'
- Detecting and guaranteeing globally valid predicates
- Distributed Ledger Algorithms (`BitCoin` et al.), ...

Some Issues for further Studies

Complexity: Trade-off between preconditions, robustness and costs.

Correctness: Important issue for safety-critical distributed systems, but especially hard to tackle due to *State-Space explosion*.

Other System Models: Normally do not meet such favorable pre conditions, e.g.,

∗ Indirect, even non-transitive, connectivity among nodes, e.g., in mobile or very heterogenous systems

∗ Modern P2P systems: no stable structure and high churn rates

Conclusion – Distributed Algorithms

▶ **Overall Goal:** *Compensate typical deficits of distributed systems through additional algorithmic layers.*

◀ **To some extent achievable but there are limitations:**

- ◁ Asynchronous systems with unpredictable error rates are barely manageable for practical applications.
  - ◀ Erroneous message channels are hard to overcome at all.
  - ◀ Constantly crashing nodes (processes) render productive work more or less impossible.

  $\Longrightarrow$ both result in a '*trade-off*' between
  - **blocking** and even deadlocks due to long waiting periods
  - **life-locks** due to timeouts and permanent re-start of algorithms
- ◁ Algorithms often costly, esp. number of msgs to be exchanged.

Practical Distributed System Development:

∗ 'Hide' preconditions by using *Middleware* with QoS guarantees.

∗ Confine algorithms to 'stable' settings, i.e., server environments.

End of chapter VI