

# Συντονισμός και συμφωνία

Κατανεμημένα Συστήματα  
2019-2020

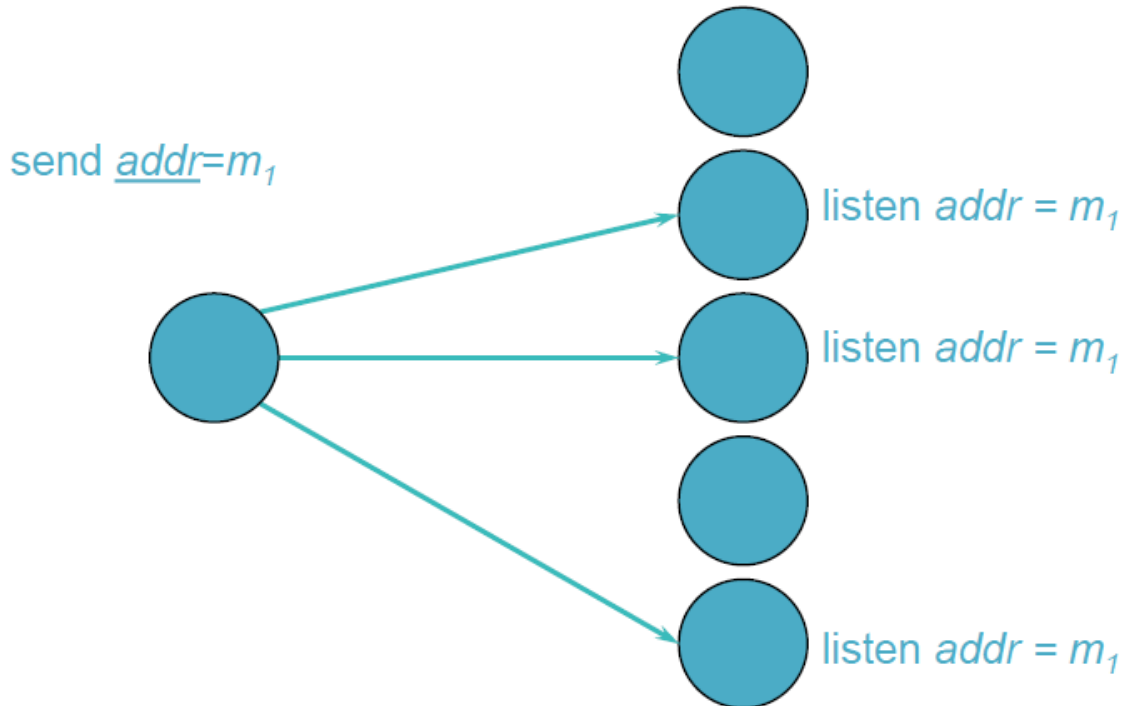
<http://www.cslab.ece.ntua.gr/courses/distrib>

# Πώς επικοινωνούν οι διεργασίες;

- Ένας προς έναν
  - Unicast
    - 1 -> 1
    - Point-to-point
- Ένας προς πολλούς
  - Multicast
    - 1 -> πολλούς
    - Επικοινωνία ομάδας (group communication)
  - Broadcast
    - 1-> όλους

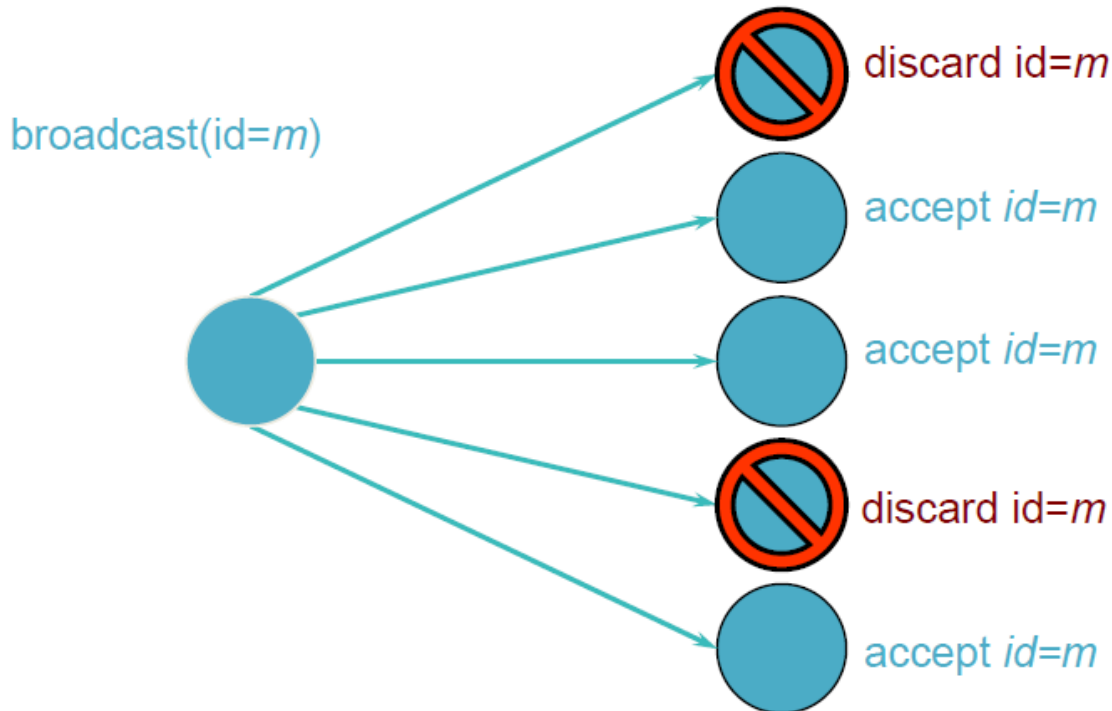
# Multicast μέσω hardware

- Αν το hardware υποστηρίζει multicast
  - Τα μέλη της ομάδας ακούν σε δικτυακές διευθύνσεις



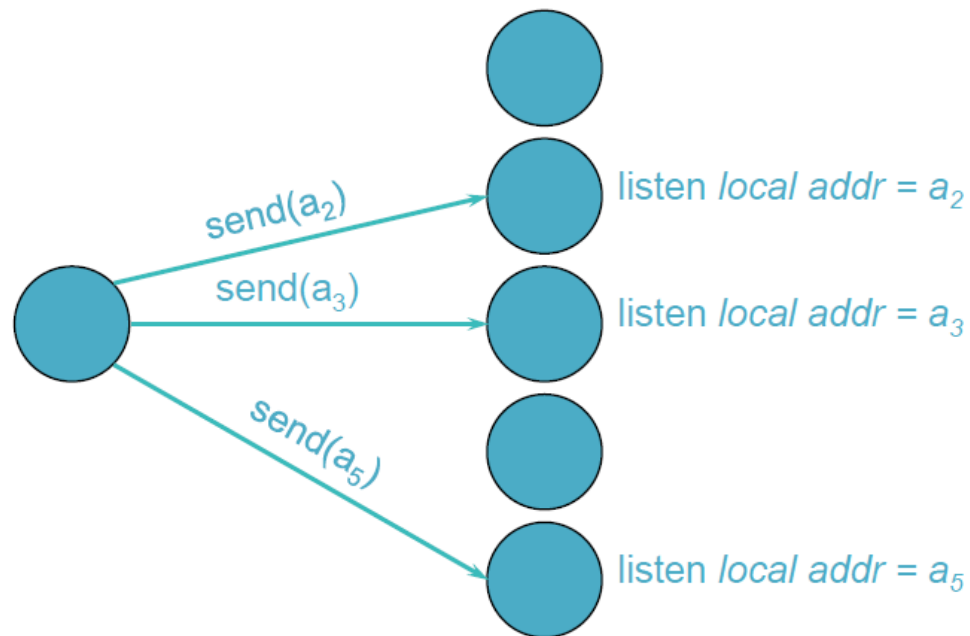
# Hardware broadcast

- Αν το hardware υποστηρίζει μόνο broadcast
  - Φιλτράρονται οι εισερχόμενες multicast διευθύνσεις μέσω λογισμικού



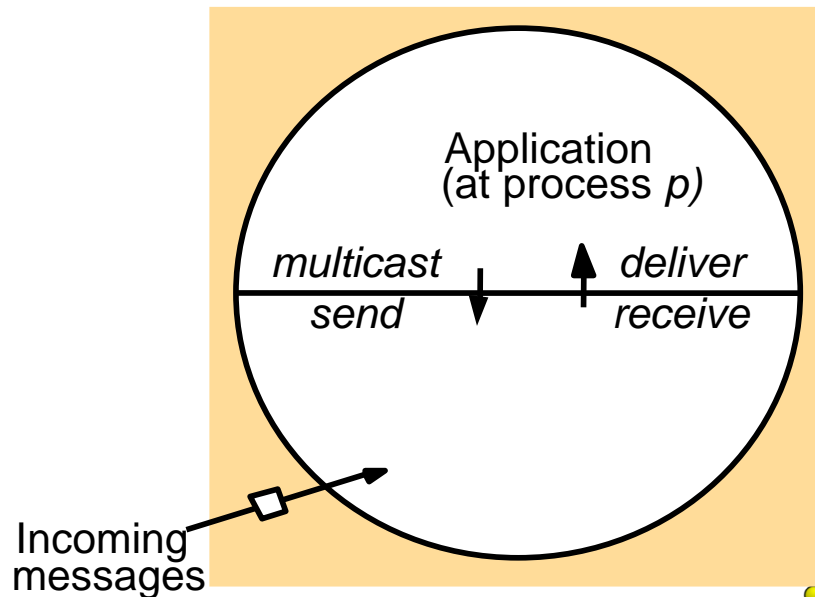
# Software multicast

- Πολλαπλά unicasts
- Ο αποστολέας γνωρίζει τα μέλη της ομάδας



# Μοντέλο συστήματος

- Multicast( $g, m$ ): στέλνει το  $m$  στην ομάδα  $g$
- Receive( $m$ ): λήψη μηνύματος
- Deliver( $m$ ): παράδοση μηνύματος στο επίπεδο εφαρμογής
- Κάθε  $m$  φέρει
  - id αποστολέα:  $sender(m)$
  - id της ομάδας παραληπτών  $group(m)$



# Βασικό multicast (B-multicast)

- Χρήση αξιόπιστου unicast
  - $B\text{-multicast}(g,m)$ : Για κάθε διεργασία  $p$  στο  $g$ ,  $\text{send}(p,m)$
  - $\text{receive}(m)$ :  $B\text{-deliver}(m)$  στο  $p$
- Εγγύηση
  - Όλες οι διεργασίες που ανήκουν στο  $g$  τελικά λαμβάνουν όλα τα multicast μηνύματα...
  - ... **αρκεί ο αποστολέας να μην αποτύχει!**

# Αξιόπιστο multicast (R-multicast)

- Όπως το B-multicast +  
όλες οι σωστές διεργασίες πρέπει να  
παραλάβουν το μήνυμα αν έστω και μια από  
αυτήν το παραλάβει
- ... γιατί ο αποστολέας μπορεί να αποτύχει ενώ  
εκτελεί B-multicast



# R-multicast: Στόχοι

- *Ακεραιότητα (integrity)*: Μια σωστή (χωρίς σφάλματα) διεργασία  $p$  παραδίδει ένα μήνυμα  $m$  το πολύ μια φορά
  - Σωστή: Τηρεί το πρωτόκολλο και είναι ζωντανή
- *Συμφωνία (agreement)*: Αν μια σωστή διεργασία παραδώσει μήνυμα  $m$ , τότε όλες οι υπόλοιπες σωστές διεργασίες στην ομάδα  $\text{group}(m)$  θα παραδώσουν τελικά το  $m$ 
  - «όλα ή τίποτα»
- *Ισχύς (validity)*: Αν μια σωστή διεργασία στείλει μήνυμα  $m$ , τότε θα παραδώσει και η ίδια το  $m$  τελικά

# R-multicast

- Διατήρηση ιστορικού μηνυμάτων για εγγύηση παράδοσης το πολύ μια φορά
- Φροντίδα να πάρουν όλοι το μήνυμα
- Αφού επιβεβαιώσουν όλοι τη λήψη, τότε ενημερώνουμε την εφαρμογή

# Αλγόριθμος

*On initialization*

*Received := {};*

*For process  $p$  to R-multicast message  $m$  to group  $g$*

*B-multicast( $g, m$ );*

*( $p \in g$  is included as destination)*

*On B-deliver( $m$ ) at process  $q$  with  $g = \text{group}(m)$*

*if ( $m \notin \text{Received}$ ):* **Integrity**

*Received := Received  $\cup$   $\{m\}$ ;*

*if ( $q \neq p$ ):*

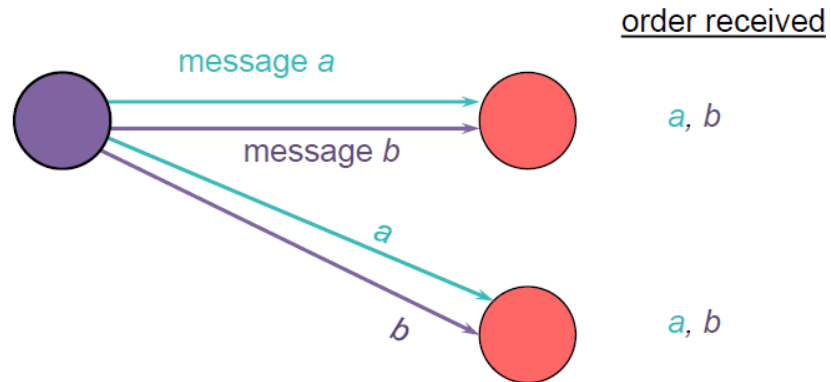
*B-multicast( $g, m$ );*

**Agreement**

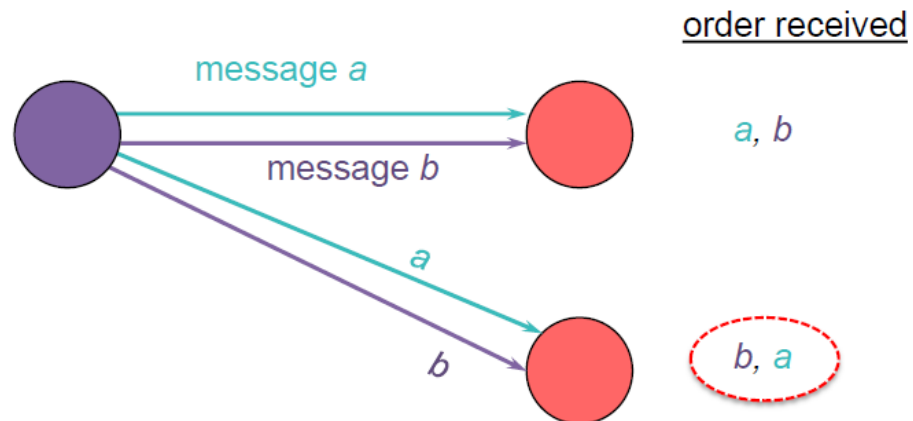
*R-deliver( $m$ )* **Validity**

# Διάταξη μηνυμάτων

- Σωστή σειρά



- Λάθος σειρά



# Διατεταγμένο multicast

- Κάθε διεργασία παραδίδει τα μηνύματα που έλαβε ανεξάρτητα από τις άλλες
- Το ερώτημα είναι, τι διάταξη ακολουθεί η κάθε διεργασία
- Τρεις τύποι διάταξης
  - FIFO
  - Αιτιώδης
  - Ολική

# Διάταξη FIFO

- Η διάταξη της παράδοσης των μηνυμάτων ακολουθεί τη σειρά αποστολής τους από κάθε διεργασία
- Αν μια σωστή διεργασία στείλει  $\text{multicast}(g,m)$  και μετά  $\text{multicast}(g,m')$ , τότε κάθε σωστή διεργασία που παραδίδει το  $m'$  θα έχει παραδώσει ήδη το  $m$
- Παράδειγμα:
  - P1:  $m_0, m_1, m_2$
  - P2:  $m_3, m_4, m_5$
  - P3:  $m_6, m_7, m_8$
- FIFO ->  $(m_0, m_3, m_6, m_1, m_4, m_7, m_2, m_5, m_8)$

# Ολική διάταξη

- Κάθε διεργασία παραδίδει όλα τα μηνύματα με την ίδια σειρά
- Αν μια σωστή διεργασία παραδώσει το μήνυμα  $m$  πριν το  $m'$  (ανεξάρτητα από τους αποστολείς), τότε οποιαδήποτε άλλη σωστή διεργασία που παραδίδει το  $m'$  θα έχει παραδώσει ήδη το  $m$
- Παράδειγμα
  - P1:  $m_0, m_1, m_2$
  - P2:  $m_3, m_4, m_5$
  - P3:  $m_6, m_7, m_8$
- Ολική διάταξη
  - P1:  $m_7, m_1, m_2, m_4, m_5, m_3, m_6, m_0, m_8$
  - P2:  $m_7, m_1, m_2, m_4, m_5, m_3, m_6, m_0, m_8$
  - P3:  $m_7, m_1, m_2, m_4, m_5, m_3, m_6, m_0, m_8$

# Αιτιώδης διάταξη (causal)

- Η διάταξη της παράδοσης των μηνυμάτων σε κάθε διεργασία διατηρεί τις σχέσεις happened-before ανάμεσα σε όλες τις διεργασίες
- Αν  $\text{multicast}(g,m) \rightarrow \text{multicast}(g,m')$  τότε κάθε σωστή διεργασία που παραδίδει το  $m'$  θα έχει παραδώσει ήδη το  $m$
- Παράδειγμα
  - P1:  $m_0, m_1, m_2$
  - P2:  $m_3, m_4, m_5$
  - P3:  $m_6, m_7, m_8$
  - Σχέσεις happened-before:  $m_0 \rightarrow m_4, m_5 \rightarrow m_8$
- Causal  $\rightarrow (m_0, m_3, m_6, m_1, m_4, m_7, m_2, m_5, m_8)$



# Παράδειγμα

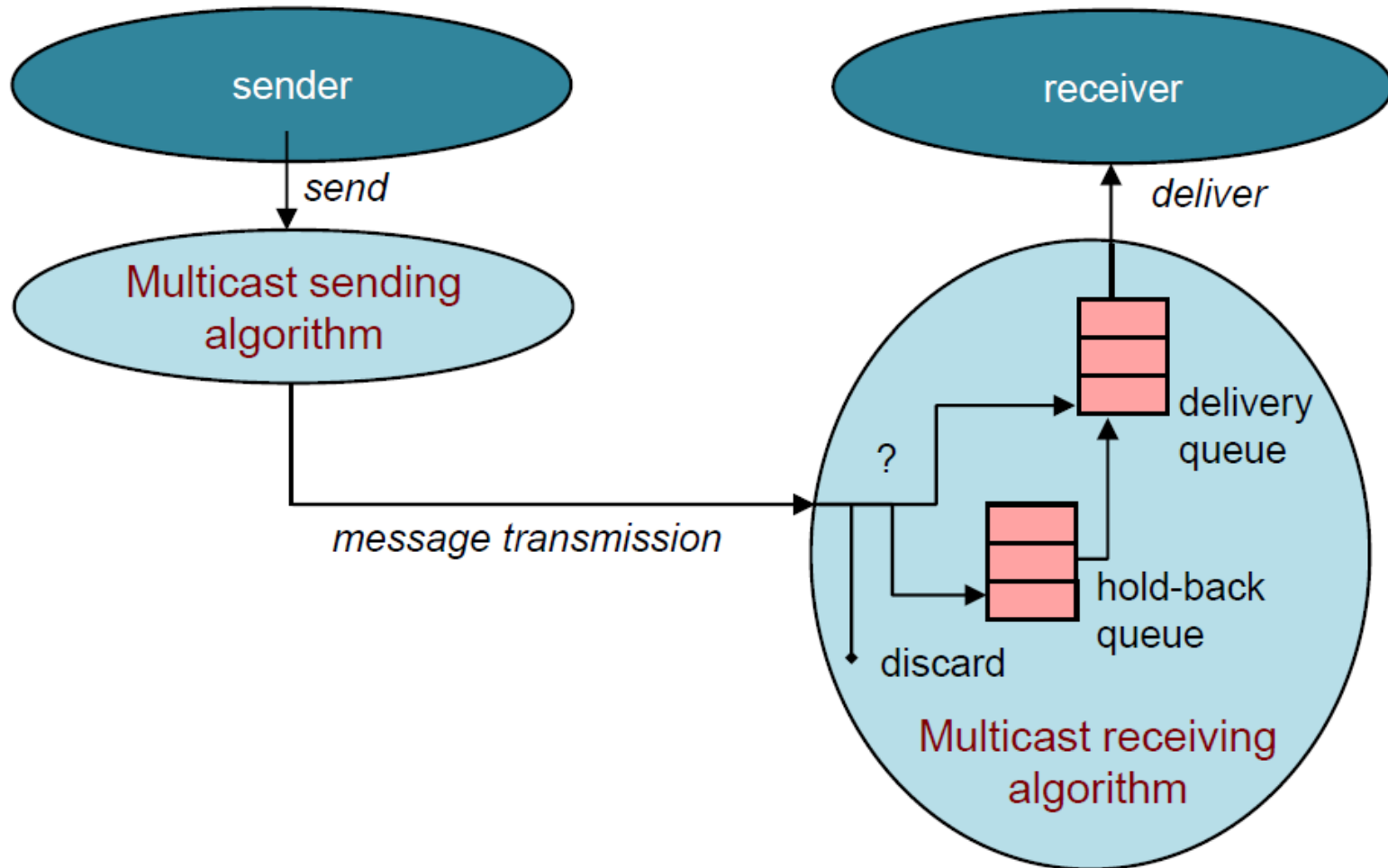
Bulletin board: <i>os.interesting</i>		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

- Αξιόπιστο Multicast : Κάθε χρήστης λαμβάνει όλα τα μηνύματα
- FIFO: Τα μηνύματα του ίδιου χρήστη εμφανίζονται με τη σειρά αποστολής
- Αιτιώδης διάταξη: Τα Re: μηνύματα μετά από αυτά στα οποία αναφέρονται
- Ολική διάταξη: Η αρίθμηση συνεπής

# Αποστολή vs. Παράδοση

- Ο αλγόριθμος λήψης ενός multicast αποφασίζει πότε να παραδώσει ένα μήνυμα στη διεργασία
- Ένα μήνυμα που παραλαμβάνεται μπορεί:
  - **Να παραδοθεί αμέσως** (να τοποθετηθεί σε ουρά παράδοσης την οποία διαβάζει η διεργασία)
  - **Να τοποθετηθεί σε ουρά αναμονής (hold-back queue)** γιατί πρέπει να περιμένουμε για παλιότερο μήνυμα
  - **Να απορριφθεί** (διπλό ή παλιότερο μήνυμα που δε θέλουμε πια)

# Αποστολή, παράδοση, αναμονή



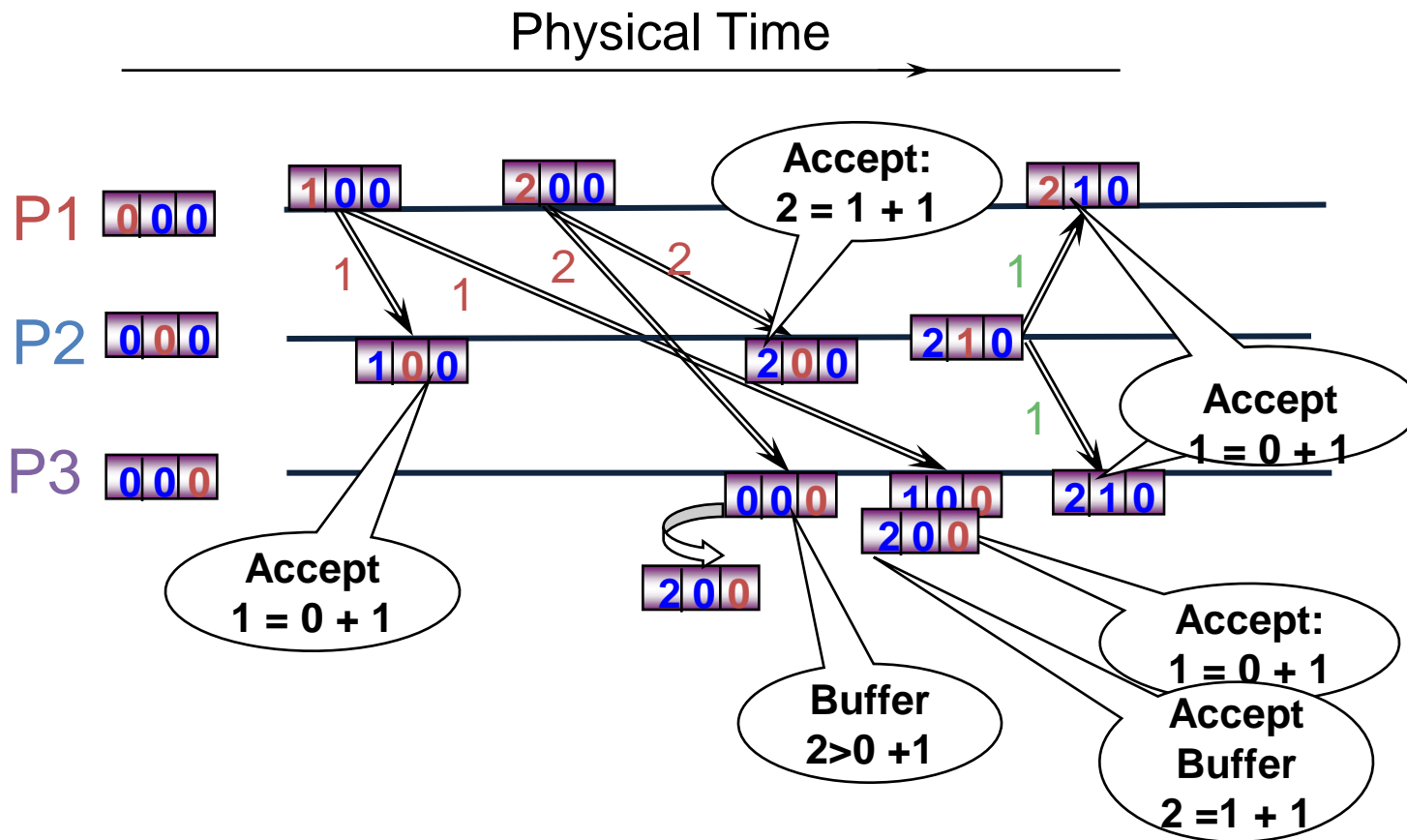
# Διάταξη FIFO

- Εξέταση μηνυμάτων από κάθε διεργασία με τη σειρά με την οποία εστάλησαν
  - Κάθε διεργασία κρατά έναν αύξοντα αριθμό για κάθε μια από τις άλλες διεργασίες
  - Όταν παραλαμβάνεται ένα μήνυμα, αν ο αύξων αριθμός του είναι:
    - Ο αναμενόμενος (επόμενος στη σειρά), το αποδεχόμαστε
    - Μεγαλύτερος από τον αναμενόμενο, το τοποθετούμε σε ουρά
    - Μικρότερος από τον αναμενόμενο, το απορρίπτουμε

# Παράδειγμα: FIFO Multicast

(do **NOT** be confused with vector timestamps)

“Accept” = Deliver



000 Sequence Vector

# Ολικά διατεταγμένο multicast

- Με τη χρήση sequencer
  - Ένας ειδικό sequencer διατάσσει όλα τα μηνύματα
  - Όλοι οι υπόλοιποι ακολουθούν
- Το σύστημα ISIS
  - Παρόμοιο με τον sequencer, αλλά η ευθύνη κατανέμεται σε όλους τους αποστολείς

# Αλγόριθμος με χρήση sequencer

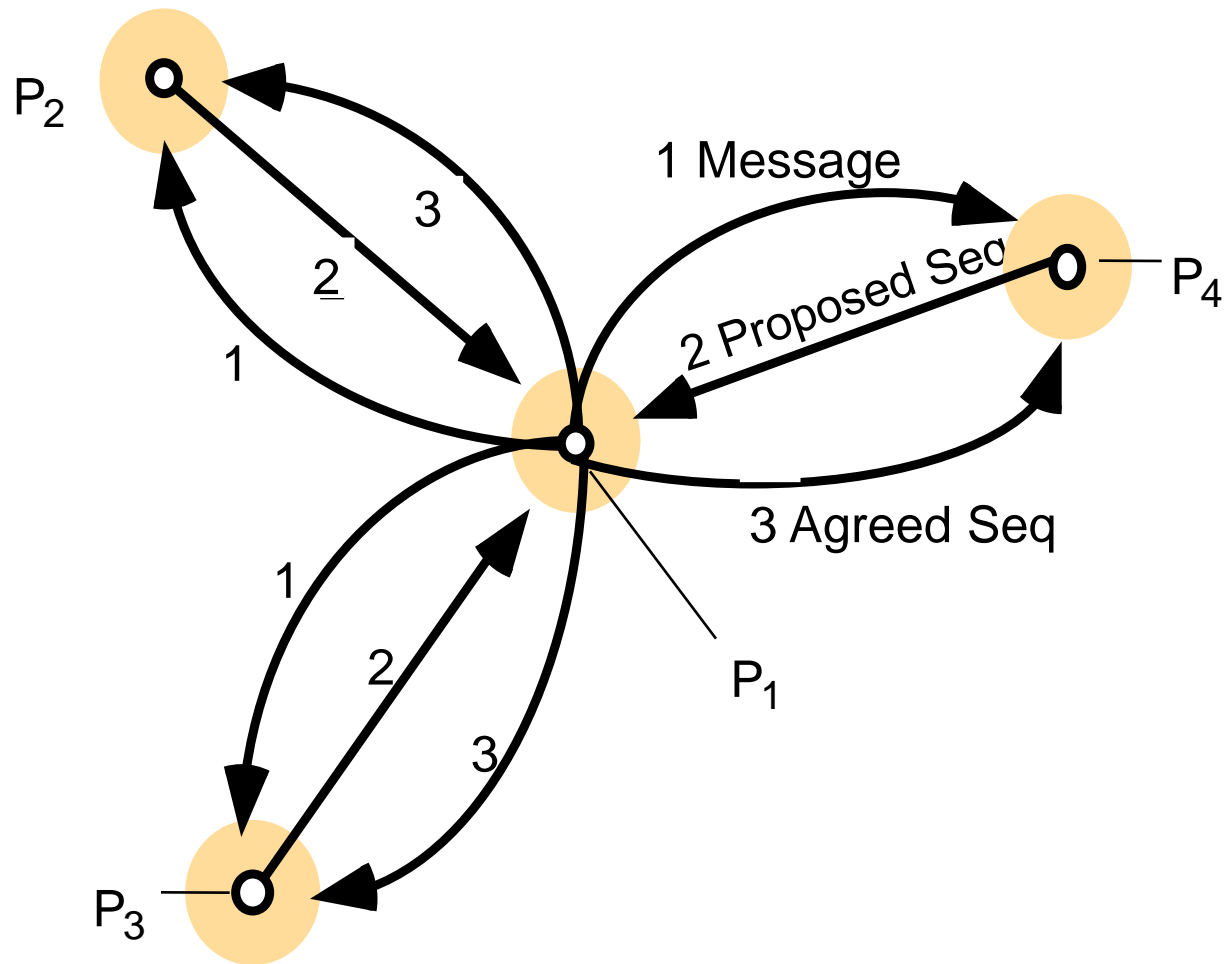
- Κάθε μήνυμα  $m$  που αποστέλλεται με multicast φέρει ένα μοναδικό id
- Το μήνυμα αποστέλλεται σε όλες της διεργασίες της ομάδα και στον sequencer
- Ο sequencer
  - κρατά έναν αύξοντα αριθμό (sequence number) που χρησιμοποιεί για να δίνει αυξανόμενους και συνεχόμενους αριθμούς στα μηνύματα
  - Ανακοινώνει τους sequence numbers στέλνοντας με multicast μήνυμα order
- Τα μηνύματα παραμένουν στην ουρά μέχρι να μπουν σε σειρά τα sequence numbers τους

# Αλγόριθμος ISIS

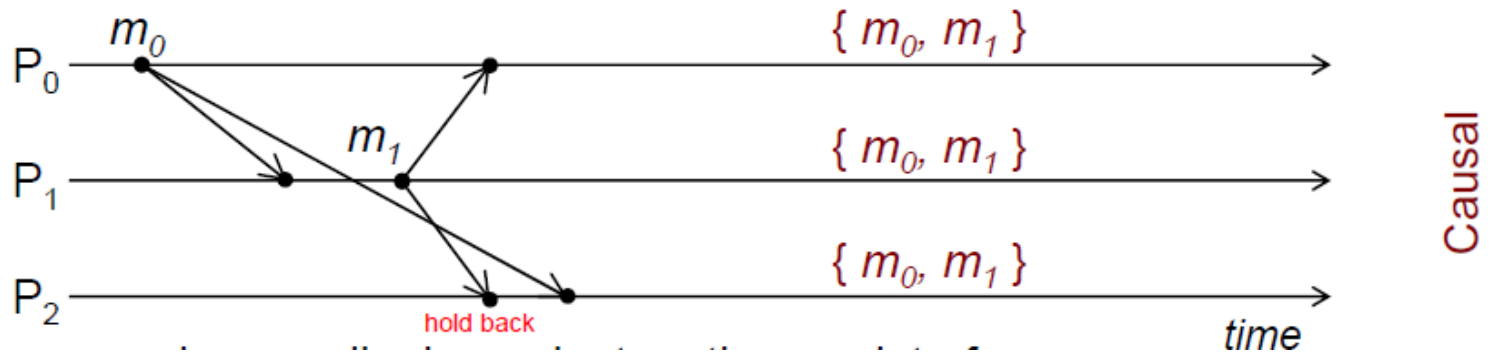
- Οι διεργασίες συμφωνούν συλλογικά σε sequence number για κάθε μήνυμα
- Οι διεργασίες απαντούν σε ένα multicast με μια πρόταση για το sequence number
  - Το μέγιστο όλων των συμφωνημένων sequence numbers
  - Μεγαλύτερο από όλα τα sequence numbers που είχαν προτείνει οι ίδιες στο παρελθόν
- Αποθηκεύουν το μήνυμα σε ουρά προτεραιότητας hold-back
  - Μηνύματα διατεταγμένα ανά sequence number
- Ο αποστολέας επιλέγει τη συμφωνημένη προτεραιότητα και ξαναστέλνει multicast με αυτήν
  - το μέγιστο όλων των προτεινόμενων προτεραιοτήτων
- Κατά τη λήψη της τελικής προτεραιότητας
  - Το μήνυμα μαρκάρεται προς παράδοση
  - Παραδίδονται μηνύματα προς παράδοση από την αρχή της ουράς προτεραιότητας



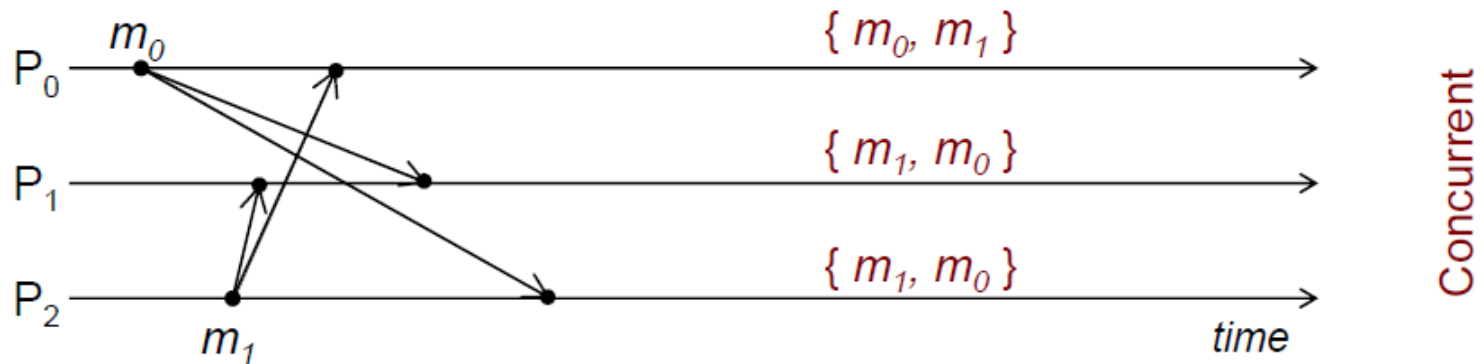
# Αλγόριθμος ISIS



# Αιτιώδης διάταξη: παράδειγμα



- Το  $m_1$  εξαρτάται από τη λήψη του  $m_0$
- Έτσι, το  $m_1$  πρέπει να παραδοθεί μετά την παράδοση του  $m_0$



- Το  $m_0$  και το  $m_1$  δεν έχουν αιτιώδη σχέση (ταυτόχρονα).
- Οποιαδήποτε διεργασία μπορεί να τα παραδώσει σε οποιαδήποτε σειρά

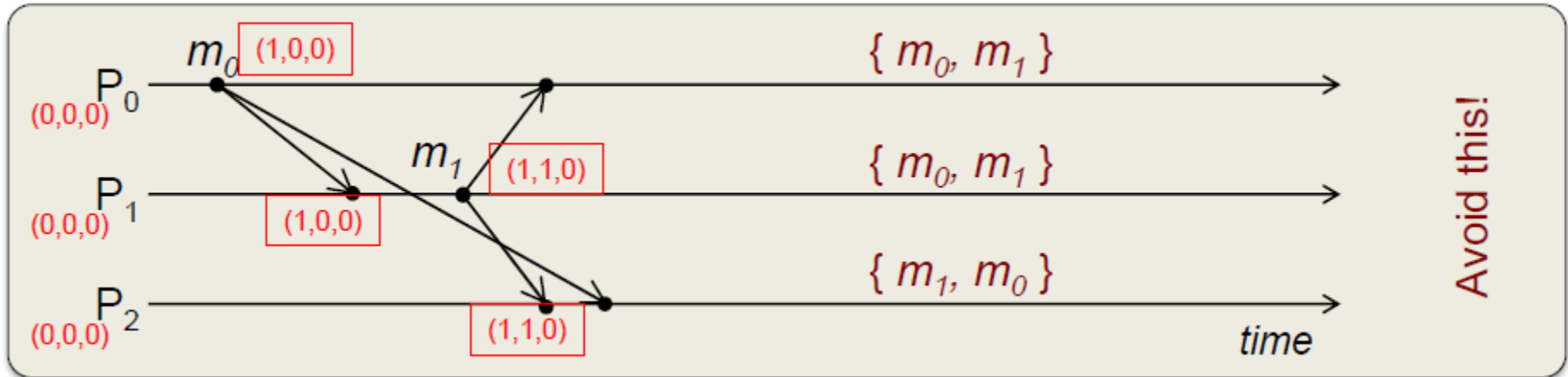
# Αιτιώδης διάταξη

- Κάθε διεργασία διατηρεί διανυσματικό ρολόι
  - Κάθε μετρητής αναπαριστά τον αριθμό των μηνυμάτων που έχουν ληφθεί από κάθε άλλη διεργασία
- Κατά το multicast, η διεργασία-αποστολέας αυξάνει τον μετρητή της και επισυνάπτει το διάνυσμα στο μήνυμα
- Κατά τη λήψη ενός μηνύματος multicast ο παραλήπτης περιμένει μέχρι να ικανοποιηθεί η αιτιώδης διάταξη, δηλ:
  - Να έχει παραδώσει όλα τα μηνύματα που είχε στείλει ο αποστολέας στο παρελθόν
  - Να έχει παραδώσει όλα τα μηνύματα που είχε παραδώσει ο αποστολέας πριν την αποστολή του multicast

# Αλγόριθμος

- Η διεργασία  $P_a$  λαμβάνει μήνυμα από την  $P_b$ 
  - Κάθε διεργασία διατηρεί ένα διάνυσμα προτεραιότητας (**παρόμοιο με διανυσματική χρονοσφραγίδα**)
  - Το διάνυσμα ανανεώνεται με κάθε αποστολή και λήψη multicast
    - Κάθε τιμή = # του πιο πρόσφατου μηνύματος από τη διεργασία που προηγείται του event
- Αλγόριθμος
  - Όταν η  $P_b$  στέλνει μήνυμα
    - Αυξάνει τη δική του τιμή και στέλνει το διάνυσμα  $V_b[b] = V_b[b] + 1$  με το μήνυμα
  - Όταν η  $P_a$  **λάβει μήνυμα από την  $P_b$** 
    - Τσεκάρει αν το μήνυμα ήρθε με σειρά FIFO από την  $P_b$   
 $V_b[b] == V_a[b] + 1$  ?
    - Τσεκάρει ότι το μήνυμα δεν εξαρτάται από κάτι που δεν έχει δει ακόμα η  $P_a$   
 $\forall i, i \neq b: V_b[i] \leq V_a[i]$  ?
    - Αν ικανοποιούνται και οι δύο συνθήκες, το  $P_a$  παραδίδει το μήνυμα
    - Αλλιώς το κρατά σε αναμονή μέχρι να ικανοποιηθούν οι συνθήκες

# Παράδειγμα



- Η  $P_2$  λαμβάνει μήνυμα  $m_1$  από την  $P_1$  με  $V_1=(1,1,0)$

(1) Είναι σε σειρά FIFO από την  $P_1$ ?

Σύγκριση τοπικού διανύσματος στην  $P_2$ :  $V_2=(0,0,0)$  με το διάνυσμα του μηνύματος από την  $P_1$ ,  $V_1=(1,1,0)$

Ναι:  $V_2[1] = 0$ ,  $V_1[1] = 1 \Rightarrow$  sequential order

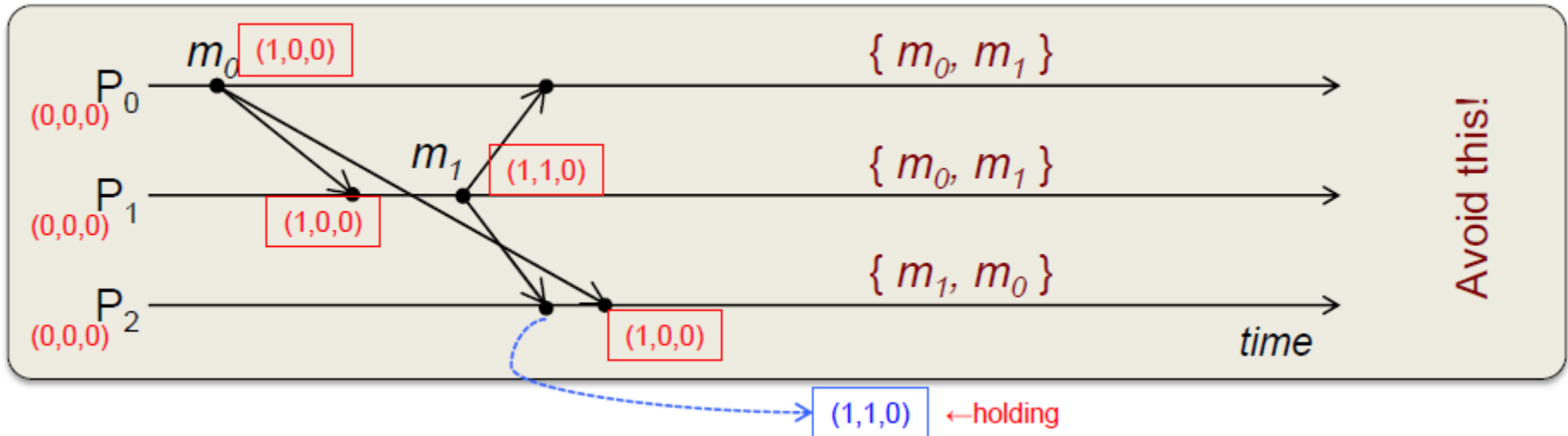
(2) Είναι  $V_1[i] \leq V_2[i]$  για όλα τα υπόλοιπα  $i$ ?

Σύγκριση των διανυσμάτων :  $V_2=(0,0,0)$  vs.  $V_1=(1,1,0)$

Νο.  $V_1[0] > V_2[0]$  ( $1 > 0$ )

Οπότε  $m_1$  σε αναμονή στην  $P_2$

# Παράδειγμα



- Η P2 λαμβάνει το  $m_0$  από την P0 με  $V=(1,0,0)$

(1) Είναι σε σειρά FIFO από την P0?

Σύγκριση τοπικού διανύσματος στην P2:  $V_2=(0,0,0)$  με το  $V_0=(1,0,0)$  από την P0

Ναι:  $V_2[0] = 0, V_0[0] = 1 \Rightarrow$  sequential

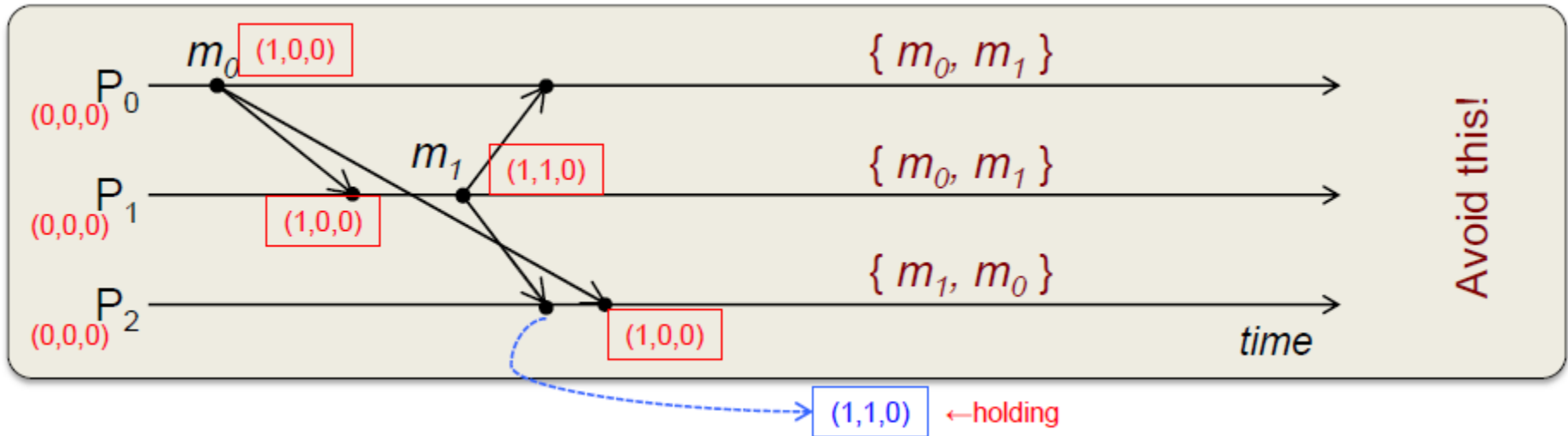
(2) Είναι  $V_0[i] \leq V_2[i]$  για όλα τα υπόλοιπα  $i$ ?

Ναι,  $(0 \leq 0), (0 \leq 0)$ .

**Παράδοση  $m_0$ .**

Τώρα τσέκαρε την ουρά αναμονής. Μπορεί να παραδοθεί το  $m_1$ ?

# Παράδειγμα



(1) Είναι το  $m_1$  που βρίσκεται σε αναμονή σε σειρά FIFO ?

Σύγκριση του τωρινού  $V$  στην  $P_2$ :  $V_2=(1,0,0)$  με το  $V$  του μηνύματος σε αναμονή από την  $P_1$ ,  $V_1=(1,1,0)$

Ναι:  $V_2[1] = 0$ ,  $V_1[1] = 1 \Rightarrow$  sequential

• (2) Είναι  $V_1[i] \leq V_2[i]$  για όλα τα υπόλοιπα  $i$ ?

Τώρα ναι. Στοιχείο 0:  $(1 \leq 1)$ , στοιχείο 2:  $(0 \leq 0)$

Παράδοση  $m_1$

- Πιο αποδοτικό από ολική διάταξη
- Δεν υπάρχει ανάγκη για sequencer.
- Δεν υπάρχει ανάγκη για acks

# Κατανεμημένος συντονισμός

- Τεχνικές για συντονισμό της εκτέλεσης ανάμεσα σε διεργασίες
- Μία διεργασία μπορεί να πρέπει να περιμένει μια άλλη
- Κοινός πόρος (κρίσιμο τμήμα) απαιτεί αποκλειστική προσπέλαση



# Γιατί θέλουμε αμοιβαίο αποκλεισμό

- Οι servers της τράπεζάς σας είναι στο cloud. Γίνονται δύο ταυτόχρονες καταθέσεις των \$1.000 στον λογαριασμό σας από διαφορετικά ATMs.
  - Και τα δύο ATMs διαβάζουν το αρχικό ποσό των \$1.000 από τον server της τράπεζας
  - Και τα δύο ATMs προσθέτουν \$1.000 στο ποσό αυτό (τοπικά στο κάθε ATM)
  - Και τα δύο γράφουν το τελικό ποσό στον server
  - Τι θα γίνει;

# Γιατί θέλουμε αμοιβαίο αποκλεισμό

- Τα ATMs πρέπει να έχουν αμοιβαία αποκλειόμενη προσπέλαση στις πληροφορίες του λογαριασμού στον server (ή στον κώδικα που αλλάζει τιμές δεδομένων στον λογαριασμό)

# Αμοιβαίος αποκλεισμός

- Κρίσιμο τμήμα
  - Ένα κομμάτι κώδικα (σε όλους τους clients) το οποίο πρέπει να εκτελείται από έναν το πολύ client κάθε στιγμή
- Λύση:
  - Σημαφόροι, mutex, κλπ. σε κεντρικά συστήματα
  - Πρωτόκολλα βασισμένα στην ανταλλαγή μηνυμάτων σε κατακεντρωμένα συστήματα:
    - enter() είσοδος στο κρίσιμο τμήμα
    - AccessResource() προσπέλαση του πόρου
    - exit() έξοδος από το κρίσιμο τμήμα
- Απαιτήσεις για κατακεντρωμένο αμοιβαίο αποκλεισμό:
  - **Safety** – Το πολύ μία διεργασία μπορεί να εκτελεί το κρίσιμο τμήμα σε κάθε χρονική στιγμή
  - **Liveness** – Κάθε αίτημα για εκτέλεση του κρίσιμου τμήματος τελικά ικανοποιείται
  - **Διάταξη** – Τα αιτήματα ικανοποιούνται με τη σειρά που γίνονται

# Κεντρικό σύστημα: mutex

- Για τον συγχρονισμό της πρόσβασης σε κοινές δομές δεδομένων από πολλαπλά νήματα

Δύο διεργασίες μπορούν να εκτελέσουν τα παρακάτω:

```
lock()
```

```
while true:                               // each iteration atomic
```

```
    if lock not in use:
```

```
        label lock in use
```

```
        break
```

```
unlock()
```

```
label lock not in use
```

# Χρήση mutex

```
mutex L= UNLOCKED;
```

ATM1:

```
lock(L); // είσοδος στο
        // κρίσιμο τμήμα
διαβάζω το υπόλοιπο του
λογαριασμού μου;
προσθέτω την κατάθεση;
ενημερώνω το υπόλοιπο;
unlock(L); // έξοδος
```

```
extern mutex L;
```

ATM2:

```
lock(L); // είσοδος στο
        // κρίσιμο τμήμα
διαβάζω το υπόλοιπο του
λογαριασμού μου;
προσθέτω την κατάθεση;
ενημερώνω το υπόλοιπο;
unlock(L); // exit
```

# Σε κατανεμημένα συστήματα

- Υποθέτουμε ότι υπάρχει συμφωνία για το πώς αναφερόμαστε σε έναν πόρο
  - Περνάει το αναγνωριστικό μαζί με τα αιτήματα (*lock("printer")*, *lock("table:employees")* κλπ)
- Σκοπός
  - Δημιουργία αλγορίθμου που επιτρέπει σε μια διεργασία να επεξεργαστεί ένα αίτημα και
  - Να αποκτήσει αποκλειστική πρόσβαση σε έναν πόρο που είναι διαθέσιμος στο δίκτυο

# Υποθέσεις

- Για όλους τους αλγορίθμους που θα εξετάσουμε κάνουμε τις ακόλουθες υποθέσεις:
  - Κάθε ζεύγος διεργασιών συνδέεται μέσω αξιόπιστου καναλιού (π.χ. TCP).
  - Τα μηνύματα παραδίδονται στους παραλήπτες με FIFO διάταξη
  - Οι διεργασίες δεν αποτυγχάνουν

# Κριτήρια Επίδοσης

- **Bandwidth:** Ο συνολικός αριθμός μηνυμάτων που αποστέλλονται σε κάθε εισαγωγή και έξοδο από το κρίσιμο τμήμα
- **Client delay:** Η καθυστέρηση που προκαλείται από μια διεργασία σε κάθε εισαγωγή ή έξοδο από το κρίσιμο τμήμα (όταν δεν υπάρχει άλλη διεργασία που να εκτελεί το κρίσιμο τμήμα ή να βρίσκεται σε αναμονή)
- **Synchronization delay:** Το χρονικό διάστημα ανάμεσα στην έξοδο μιας διεργασίας από το κρίσιμο τμήμα και την εισαγωγή της επόμενης διεργασίας σε αυτό (όταν υπάρχει μόνο μια διεργασία σε αναμονή).



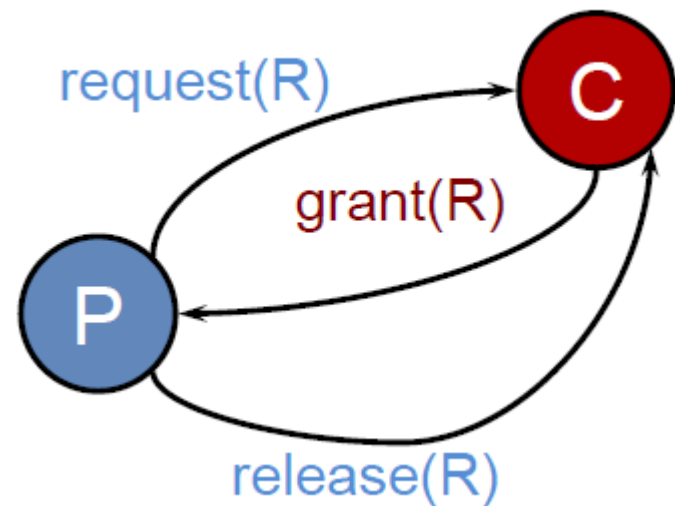
# Κατηγορίες αλγορίθμων

- Κεντρικοί
  - Μια διεργασία αποκτά πρόσβαση σε έναν πόρο όταν της το επιτρέπει ένας κεντρικός συντονιστής (κεντρικός έλεγχος)
- Με σκυτάλη
  - Μια διεργασία αποκτά πρόσβαση σε έναν πόρο αν έχει σκυτάλη (token) που της το επιτρέπει (Δακτύλιος με σκυτάλη)
- Με συμφωνία
  - Μια διεργασία αποκτά πρόσβαση σε έναν πόρο μέσω καταναεμημένης συμφωνίας (Ricart & Agrawala)

# Κεντρικός Έλεγχος

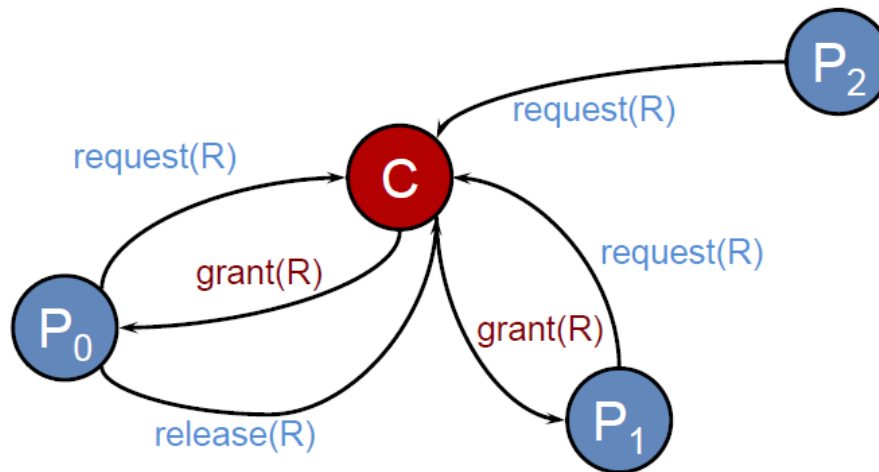
- Μια διεργασία έχει εκλεγεί ως συντονιστής (coordinator)

1. Αίτηση για πρόσβαση
2. Αναμονή για απάντηση
3. Λήψη άδειας
4. Πρόσβαση σε κρίσιμο τμήμα
5. Έξοδος



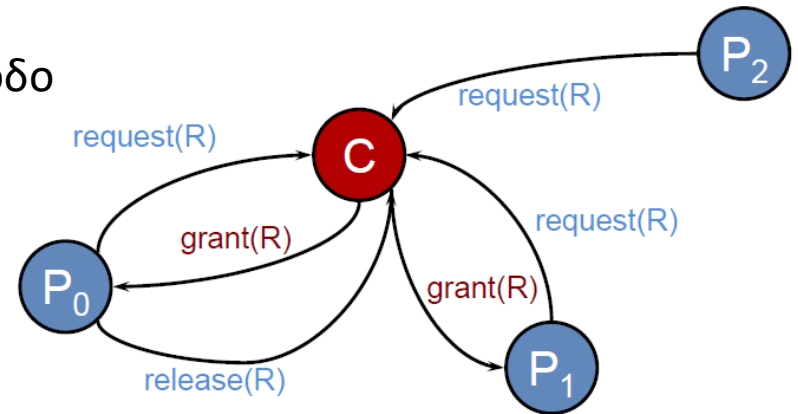
# Κεντρικός Έλεγχος

- Αν κάποια άλλη διεργασία ζητήσει τον πόρο τότε:
  - Ο συντονιστής δεν αποκρίνεται μέχρι να απελευθερωθεί ο πόρος
  - Διατηρεί ουρά
- Τα αιτήματα εξυπηρετούνται με σειρά FIFO



# Αποτίμηση αλγορίθμου

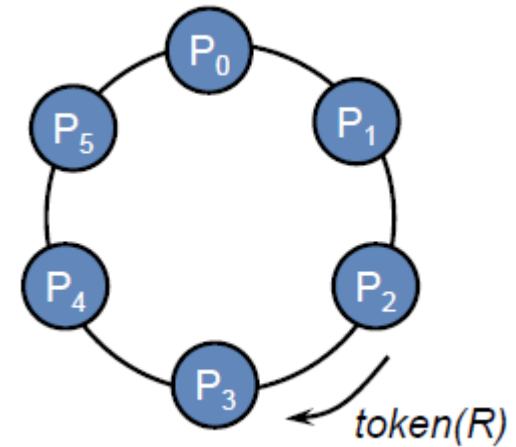
- Bandwidth
  - Απαιτούνται 3 μηνύματα ανά είσοδο/έξοδο
- Client delay
  - Ένα round trip time (αίτημα + άδεια)
- Synchronization delay
  - Ένα round trip time (έξοδος + άδεια)



- + Δικαιοσύνη: Τα αιτήματα υφίστανται επεξεργασία με τη σειρά
- + Ευκολία υλοποίησης και επαλήθευσης
- Η διεργασία δεν μπορεί να ξεχωρίσει αν μπλοκάρεται από συντονιστή που έχει σφάλμα
- Ο κεντρικός εξυπηρετητής μπορεί να αποτελέσει bottleneck

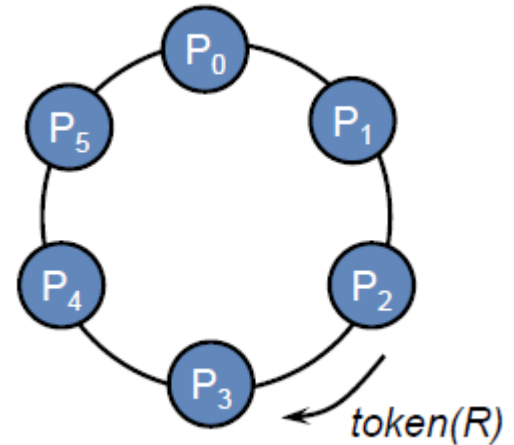
# Αλγόριθμος δακτυλίου

- Μπορεί να επιβληθεί κάποια διάταξη
- Οι διεργασίες είναι οργανωμένες σε λογικό δακτύλιο
- Κάθε διεργασία επικοινωνεί με τη γειτονική της
- Αρχικοποίηση
  - Η διεργασία 0 παίρνει σκυτάλη για τον πόρο R
- Η σκυτάλη κυκλοφορεί γύρω από τον δακτύλιο
  - Από την  $P_i$  στην  $P_{(i+1) \bmod N}$
- Όταν η διεργασία παίρνει τη σκυτάλη
  - Ελέγχει αν χρειάζεται τον πόρο
  - Αν όχι, στέλνει τη σκυτάλη στην επόμενη
  - Αν ναι, προσπελάζει τον πόρο
- Κρατά τη σκυτάλη μέχρι να ολοκληρώσει



# Αποτίμηση αλγορίθμου

- Bandwidth: 1 μήνυμα ανά έξοδο
- Client delay: 0 με N μηνύματα
- Synchronization delay: ανάμεσα σε 1 και N-1 μηνύματα
- Μόνο μια διεργασία τη φορά έχει τη σκυτάλη
  - + Ο αμοιβαίος αποκλεισμός είναι εγγυημένος
- Η διάταξη είναι καλά ορισμένη (όχι απαραίτητα FIFO)
  - + Δεν μπορεί να προκληθεί Starvation
- Αν η σκυτάλη χαθεί (π.χ., η διεργασία πεθάνει)
  - Θα πρέπει να δημιουργηθεί ξανά
  - Η ανίχνευση της απώλειας είναι δύσκολη (η σκυτάλη χάθηκε ή χρησιμοποιείται από κάποια άλλη διεργασία;)



# Αλγόριθμος Ricart & Agrawala

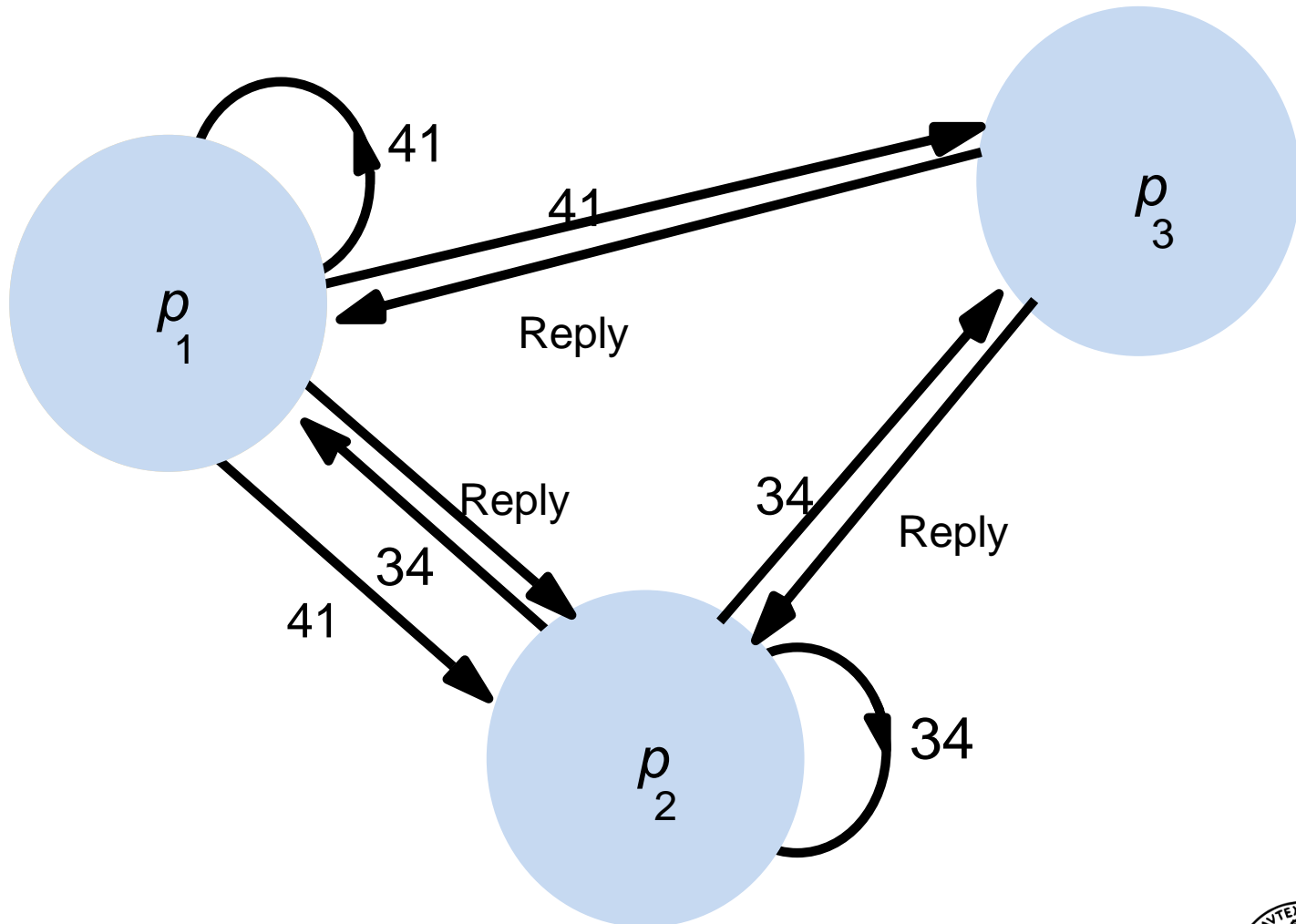
- Κατανεμημένος αλγόριθμος που χρησιμοποιεί αξιόπιστο multicast και λογικά ρολόγια
- Όταν μια διεργασία θέλει να εισέλθει στο κρίσιμο τμήμα:
  1. Συνθέτει μήνυμα που περιέχει:
    - Αναγνωριστικό (machine ID, process ID)
    - Όνομα του πόρου
    - Χρονοσφραγίδα (π.χ., Lamport)
  2. Στέλνει με αξιόπιστο multicast αίτημα σε όλες της διεργασίες της ομάδας
  3. Περιμένει μέχρι να λάβει άδεια από όλους
  4. Εισέρχεται στο κρίσιμο τμήμα/χρησιμοποιεί τον πόρο

# Συνέχεια

- Για να εισέλθει στο κρίσιμο τμήμα
  - θέτει την κατάστασή του σε wanted
  - στέλνει με multicast αίτημα σε όλες τις διεργασίες (συμπεριλαμβάνοντας χρονοσφραγίδα)
  - Περιμένει μέχρι όλες οι διεργασίες να στείλουν απάντηση
  - Αλλάζει την κατάστασή του σε held και μπαίνει στο κρίσιμο τμήμα
- Κατά τη λήψη αιτήματος  $\langle T_i, p_i \rangle$  στην  $p_j$ :
  - Αν η κατάστασή της είναι held ή η κατάσταση είναι wanted &  $(T_j, p_j) < (T_i, p_i)$ , βάλε το αίτημα σε ουρά
  - αλλιώς απάντησε στην  $p_i$
- Κατά την έξοδο από το κρίσιμο τμήμα
  - Άλλαξε την κατάσταση σε release και απάντησε στο πρώτο αίτημα στην ουρά



# Παράδειγμα



# Αποτίμηση

- Bandwidth:
    - $2(N-1)$  μηνύματα ανά εισαγωγή
    - $N-1$  unicasts για το multicast request +  $N-1$  απαντήσεις
  - Client delay
    - Ένα RTT
  - Synchronization delay
    - Ο χρόνος μετάδοσης ενός μηνύματος
- Πιο ακριβός αλγόριθμος σε bandwidth
- $N$  σημεία σφάλματος
- + Μικρότερο synchronization delay (1 μήνυμα)
- + Βελτιώσεις: Αν η διεργασία που μπήκε τελευταία στο critical section δεν έχει λάβει άλλα αιτήματα;

# Αλγόριθμοι Εκλογής

# Γιατί χρειάζονται;

- Όταν θέλουμε μια διεργασία να παίξει έναν συγκεκριμένο ρόλο
  - Συντονιστή για αμοιβαίο αποκλεισμό
  - Τον root σε ομάδα από NTP servers

# Παραδοχές

- Οποιαδήποτε διεργασία μπορεί να ξεκινήσει διαδικασία εκλογής
- Κάθε διεργασία μπορεί να ξεκινήσει το πολύ μια διαδικασία εκλογής τη φορά
- Πολλές διεργασίες μπορούν να ξεκινήσουν διαδικασία εκλογής ταυτόχρονα
  - Όλες πρέπει να αποφασίσουν για έναν μοναδικό, κοινό αρχηγό
  - Το αποτέλεσμα δεν εξαρτάται από το ποια διεργασία ξεκίνησε τη διαδικασία εκλογής
- Τα μηνύματα τελικά θα παραδοθούν

# Ορισμός προβλήματος

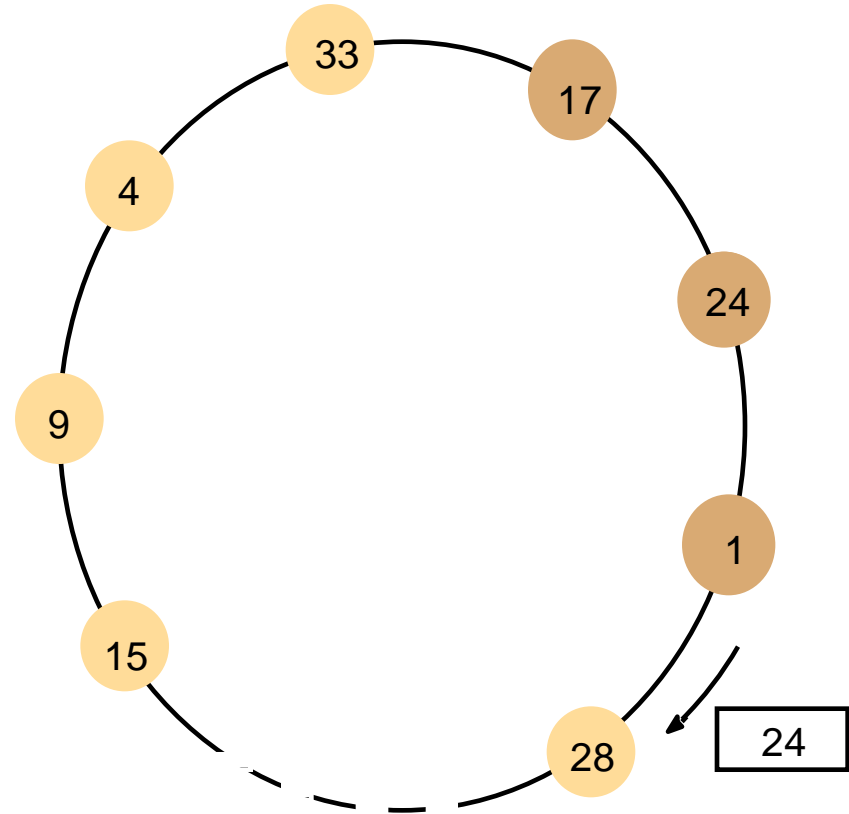
- Στο τέλος του πρωτοκόλλου εκλογής επιλέγεται ως αρχηγός η ζωντανή διεργασία με το μεγαλύτερο αναγνωριστικό
  - Παράδειγμα αναγνωριστικού: CPU speed, load, disk space, ID
  - Πρέπει να είναι μοναδικό και να μπορεί να διαταχθεί
- Κάθε διεργασία έχει μια μεταβλητή *elected*.
- Η εκτέλεση του αλγορίθμου πρέπει πάντα στο τέλος να εγγυάται
  - **Safety**:  $\forall$  σωστή  $p$ : ( *elected* = ( $q$ : μια συγκεκριμένη σωστή διεργασία με τη μεγαλύτερη τιμή αναγνωριστικού) )
  - **Liveness**:  $\forall$  election: (η διαδικασία τερματίζει) &  $\forall$  σωστή διεργασία  $p$ : η μεταβλητή *elected* του  $p$  είναι τελικά διάφορη του  $\perp$

# Εκλογή δακτυλίου

- N διεργασίες οργανώνονται σε λογικό δακτύλιο
  - η  $p_i$  επικοινωνεί με την  $p_{i+1 \bmod N}$ .
  - Όλα τα μηνύματα στέλνονται δεξιόστροφα γύρω από τον δακτύλιο
- Για την έναρξη της διαδικασίας εκλογής
  - Η διεργασία στέλνει μήνυμα *election* μαζί με το id της
- Κατά τη λήψη μηνύματος (*election*, id) μια διεργασία με myID
  - Αν  $id > my ID$ : προωθεί το μήνυμα
    - Θέτει την κατάστασή της σε *participating*
  - Αν  $id < my ID$ : στέλνει (*election*, my ID)
    - Skip αν η κατάσταση είναι ήδη *participating*
    - Θέτει την κατάσταση σε *participating*
  - Αν  $id = my ID$ : η διεργασία εξελέγη αρχηγός και στέλνει μήνυμα *elected*
    - Το μήνυμα *elected* προωθείται μέχρι να φτάσει ξανά στον αρχηγό.

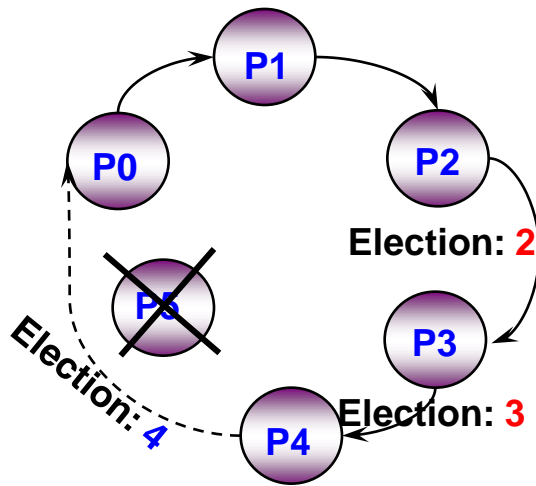
# Αποτίμηση

- Ποιο είναι το χειρότερο σενάριο;
  - Το μεγαλύτερο id έχει η διεργασία που βρίσκεται ακριβώς πριν τη διεργασία που ξεκίνησε τη διαδικασία εκλογής
  - $3N-1$  μηνύματα
- Παράδειγμα :
  - Η διαδικασία ξεκίνησε από τη διεργασία 17
  - Το μεγαλύτερο id μέχρι τώρα είναι το 24
  - Ο τελικός αρχηγός είναι η διεργασία με id 33

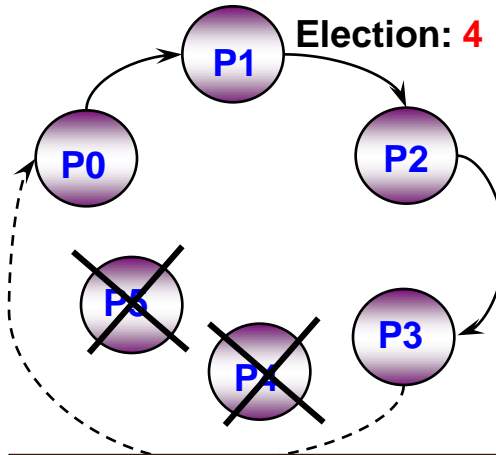




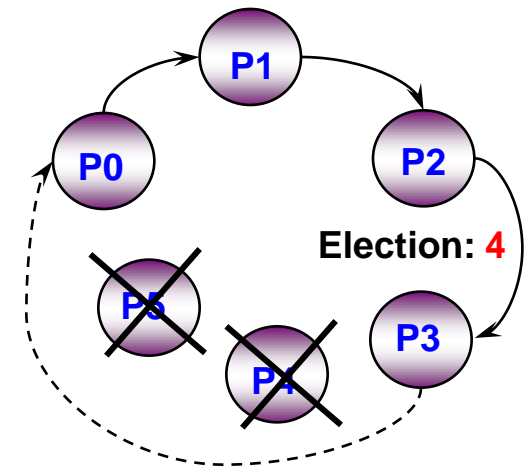
# Παράδειγμα



1. Η P2 ξεκινά εκλογή μετά την αποτυχία της P5



2. Η P2 λαμβάνει μήνυμα election και η P4 πεθαίνει



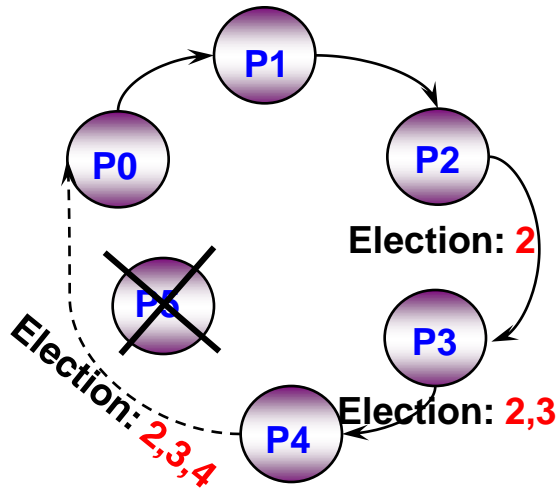
3. Το μήνυμα election: 4 προωθείται για πάντα

Αν μια διεργασία πεθάνει κατά τη διάρκεια εκτέλεσης, ο αλγόριθμος μπορεί να μην τερματίσει ποτέ

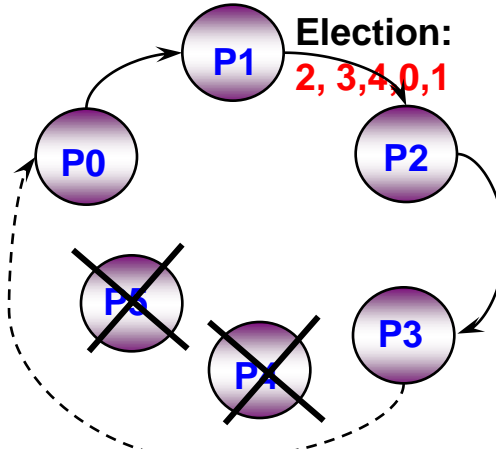
# Τροποποιημένη εκλογή δακτυλίου

- Το μήνυμα *election* καταγράφει όλα τα ids των κόμβων που το προώθησαν, όχι μόνο το μεγαλύτερο
  - Κάθε διεργασία προσθέτει το id της στη λίστα
- Μόλις το μήνυμα περάσει όλον τον κύκλο, αποστέλλεται ένα νέο μήνυμα *coordinator*
  - Επιλέγεται ο κόμβος με το μεγαλύτερο id στο μήνυμα *election*
  - Κάθε κόμβος προσθέτει το id του στο μήνυμα *coordinator*
- Όταν το μήνυμα *coordinator* επιστρέψει στον εκκινητή της διαδικασίας
  - Η εκλογή έχει επιτύχει αν το id του νέου αρχηγού είναι μέσα στη λίστα των ids
  - Αλλιώς ξεκινάει νέα διαδικασία εκλογής

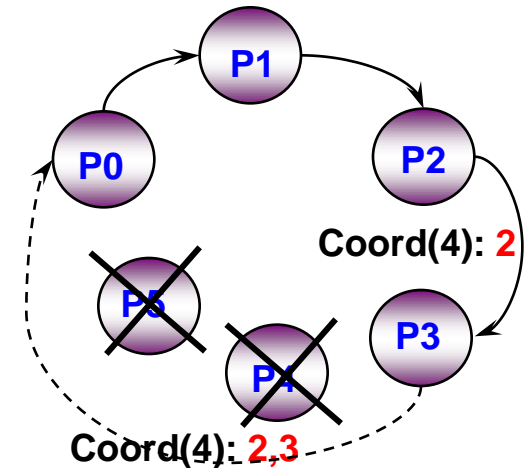
# Παράδειγμα



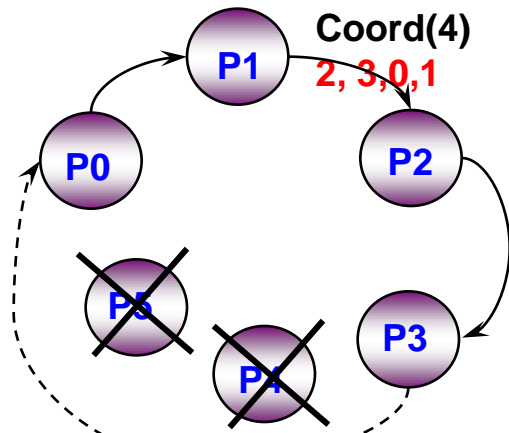
1. Η P2 ξεκινά εκλογή



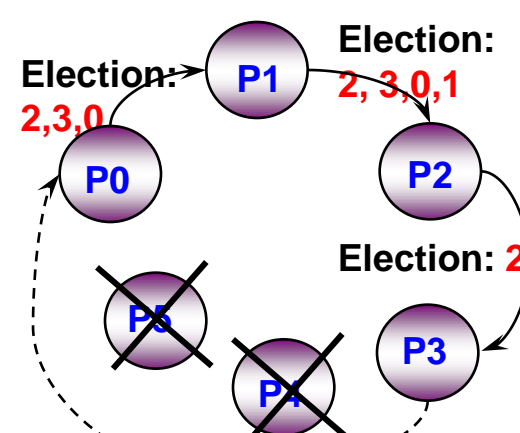
2. Η P2 λαμβάνει μήνυμα election και η P4 πεθαίνει



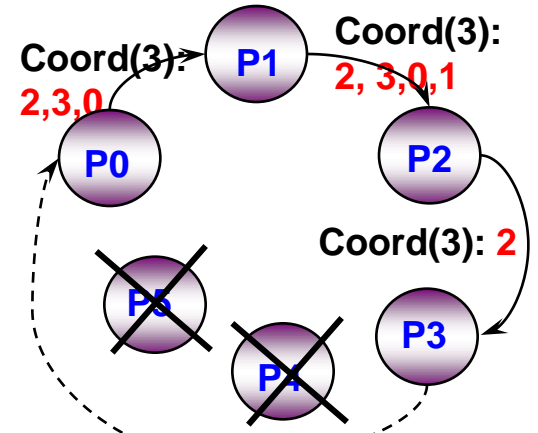
3. Η P2 επιλέγει την 4 και ανακοινώνει



4. Η P2 λαμβάνει το μήνυμα coord χωρίς το id 4



5. Η P2 ξεκινά τη διαδικασία ξανά



6. Τελικά εκλέγεται η P3

# Αποτίμηση

- Πόσα μηνύματα;
  - $2N$
  - Όμως μεγαλύτερα σε μέγεθος
- Αν πεθάνει ο εκκινητής;
  - Ο επόμενος κόμβος βλέπει ότι το μήνυμα πέρασε από όλον τον δακτύλιο
  - Ξεκινά καινούρια εκλογή

# Αλγόριθμος bully

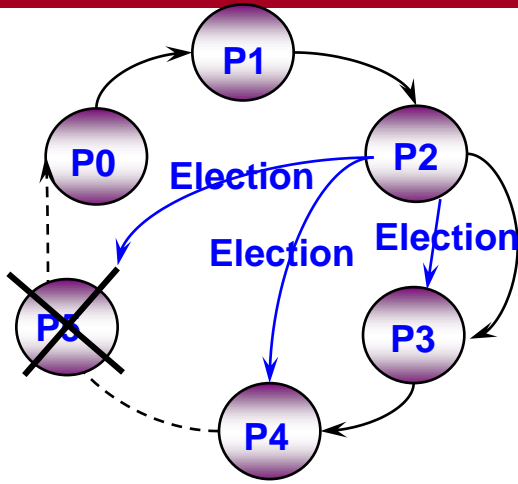
- Υποθέσεις
  - Σύγχρονο σύστημα (δλδ υπάρχουν όρια στο χρόνο μετάδοσης ενός μηνύματος)
  - Χρησιμοποιεί timeouts για να ανιχνεύσει σφάλματα σε διεργασίες
  - Κάθε διεργασία γνωρίζει όλες τις υπόλοιπες διεργασίες στο σύστημα (επομένως και τα ids τους)

# Αλγόριθμος bully

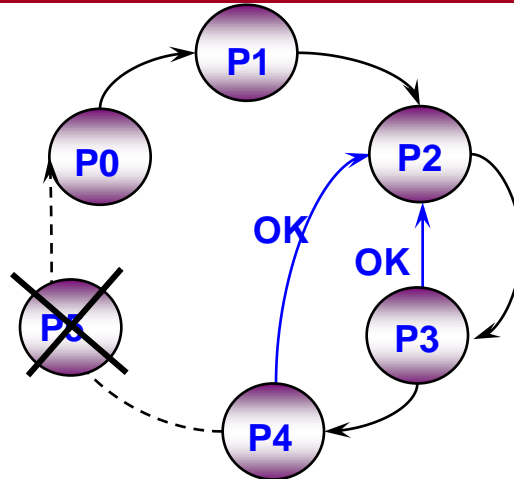
- 3 τύποι μηνυμάτων
  - *election* – εκκίνηση διαδικασίας εκλογής
  - *answer* – επιβεβαίωση της λήψης
  - *coordinator* – ανακήρυξη αρχηγού
- Έναρξη διαδικασίας
  - Ο εκκινητής στέλνει μήνυμα *election* σε όλες τις διεργασίες με id μεγαλύτερο από το δικό του
  - Αν δεν απαντήσει κανείς μέχρι το timeout: ανακηρύσσεται αρχηγός
  - Αν κάποιος απαντήσει, περιμένει για το μήνυμα *coordinator*
    - Επανεκκινεί τη διαδικασία μετά από timeout
- Κατά τη λήψη μηνύματος *election*
  - Αποστολή *answer*
  - Εκκίνηση νέας διαδικασίας εκλογής

# Παράδειγμα

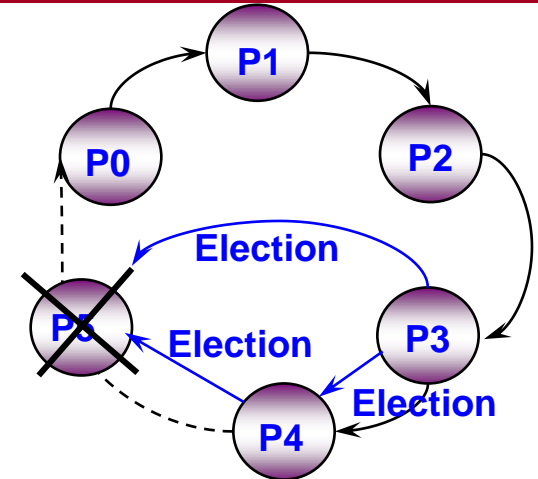
answer=OK



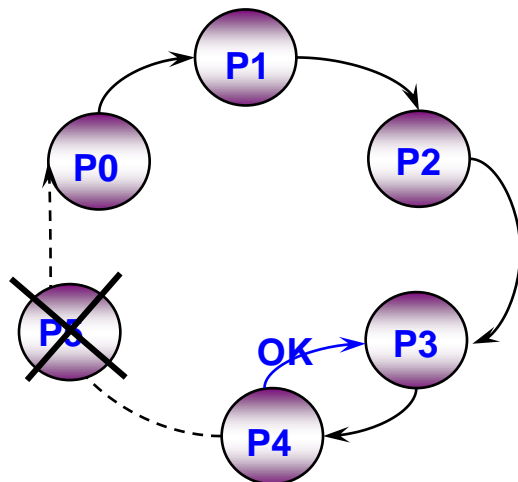
1. Η P2 ξεκινά εκλογή



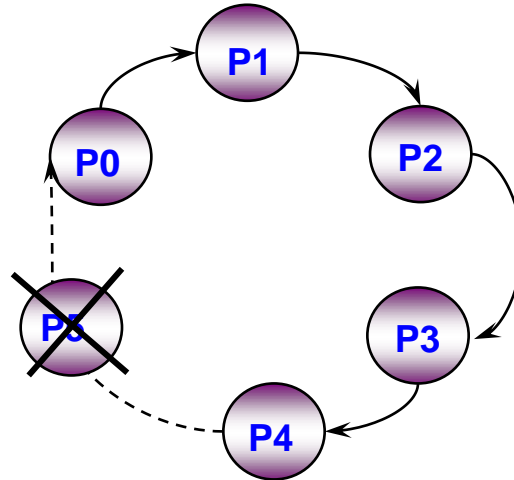
2. Η P2 λαμβάνει απάντηση



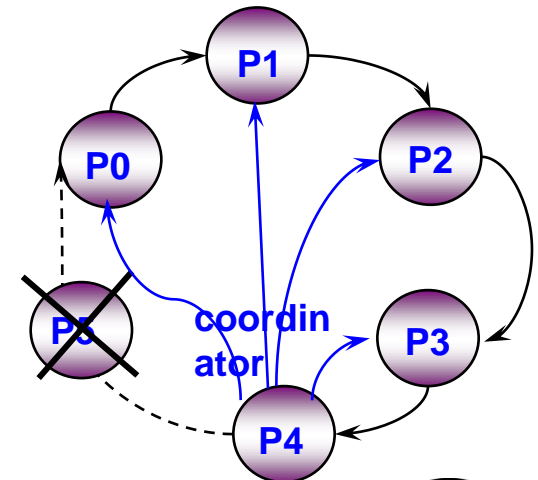
3. Οι P3 & P4 ξεκινούν εκλογή



4. Η P3 λαμβάνει απάντηση



5. Η P4 δε λαμβάνει απάντηση



5. Η P4 ανακηρύσσεται αρχηγός

# Αποτίμηση

- Best case scenario: Η διεργασία με το δεύτερο μεγαλύτερο id αντιλαμβάνεται ότι ο συντονιστής έχει πεθάνει και εκλέγει τον εαυτό της
  - N-2 μηνύματα *coordinator*
- Worst case scenario: Η διεργασία με το μικρότερο id ανιχνεύει ότι ο συντονιστής έχει πεθάνει
  - N-1 διεργασίες ξεκινούν εκλογή ταυτόχρονα, καθεμιά από τις οποίες στέλνει μηνύματα στις διεργασίες με μεγαλύτερο id
  - Η επιβάρυνση σε μηνύματα είναι  $O(N^2)$ .
- Πόσο να θέσουμε το timeout;
  - $2T + T_{\text{process}}$



# Ανακεφαλαίωση

- Group communication
  - Multicast για FIFO διάταξη
  - Multicast για ολική διάταξη
    - Sequencer
    - ISIS
  - Multicast για αιτιώδη διάταξη
    - Χρησιμοποιεί vector timestamps
- Αμοιβαίος αποκλεισμός
  - Κεντρικός έλεγχος
  - Δακτύλιος με σκυτάλη
  - Αλγόριθμος Ricart and Agrawala
- Ο συντονισμός σε καταναμημένα συστήματα απαιτεί συχνά εκλογή αρχηγού
  - Αλγόριθμος δακτυλίου
  - Αλγόριθμος τροποποιημένου δακτυλίου
  - Αλγόριθμος bully