Ρολόγια και Συγχρονισμός

Κατανεμημένα Συστήματα

2019-2020

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

Συνοπτικά

- Πρέπει να ξέρουμε πότε έγινε τι
  - Ιδανικά **ακριβώς** πότε έγινε τι
- Ή τουλάχιστον να διατάξουμε χρονικά τα γεγονότα σε διαφορετικές διεργασίες που τρέχουν ταυτόχρονα
- Συγχρονισμός ανάμεσα σε αποστολείς και παραλήπτες μηνυμάτων
- Θέματα
  - Φυσικά Ρολόγια
  - Λογικά Ρολόγια
  - Συγχρονισμός φυσικών και λογικών ρολογιών
  - Καθολικές Καταστάσεις

Παράδειγμα

- Αγορά αεροπορικών εισιτηρίων online
- Το σύστημα έχει 2 εξυπηρετητές, Α και Β
- Γιατί να μη χρησιμοποιήσουμε απλώς χρονοσφραγίδες;
  - Κάνεις κράτηση για το τελευταίο εισιτήριο
  - Ο Α σου δίνει χρονοσφραγίδα 9h:15m:32.45s
  - Κάποιος άλλος έκανε κράτηση μέσω του Β στις 9h:20m:22.76s?
  - Αν το ρολόι του Α πηγαίνει 10 λεπτά μπροστά από αυτό του Β; Πίσω;
  - Σε ποιόν θα πουληθεί τελικά το τελευταίο εισιτήριο;

Κάποιοι ορισμοί

- Κατανεμημένο σύστημα
  - Σύνολο από N processes p_i, i = 1, 2, ...N
  - Επικοινωνούν μόνο μέσω μηνυμάτων

- Event
  - Επικοινωνία (αποστολή ή λήψη μηνύματος)
  - Λειτουργία που αλλάζει την κατάσταση της p_i

Φυσικά Ρολόγια

- Κάθε υπολογιστής έχει το δικό του φυσικό ρολόι
- Skew: Διαφορά στο χρόνο μεταξύ 2 ρολογιών
- Drift: Διαφορά στη συχνότητα μεταξύ 2 ρολογιών
- Μη μηδενικό drift -> συνεχής αύξηση του skew
- UTC : Coordinated Universal Time standard
  - broadcast από εξωτερική πηγή μεγάλης ακρίβειας

Συγχρονισμός φυσικών ρολογιών

- $C_i(t)$: Η τιμή ενός software clock του process *i* όταν ο πραγματικό χρόνος είναι *t*.
- Εξωτερικός συγχρονισμός: S η πηγή UTC χρόνου και *D>0* θετικό όριο τότε αν

$$\left|S(t) - C_i(t)\right| < D,$$

για όλες τις τιμές του i και του t τότε τα ρολόγια C_i είναι ακριβή με όριο *D*

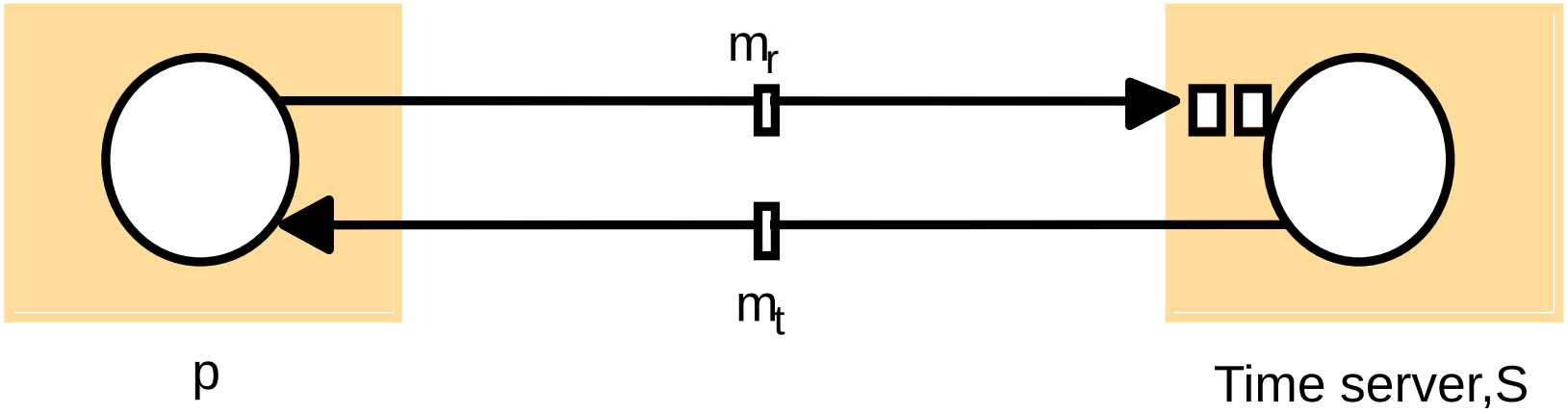
- Εσωτερικός συγχρονισμός: Για θετικό όριο *D>0*, αν

$$\left|C_i(t) - C_j(t)\right| < D$$

για όλα τα ζευγάρια i, j και τιμές του χρόνου *t,* τότε τα ρολόγια C_i, C_j συμφωνούν μέσα στο όριο *D*.

- Εξωτερικός συγχρονισμός με *D* $\Rightarrow$ Εσωτερικός συγχρονισμός με *2D*

Χρησιμοποιώντας time server



Αλγόριθμος του Cristian

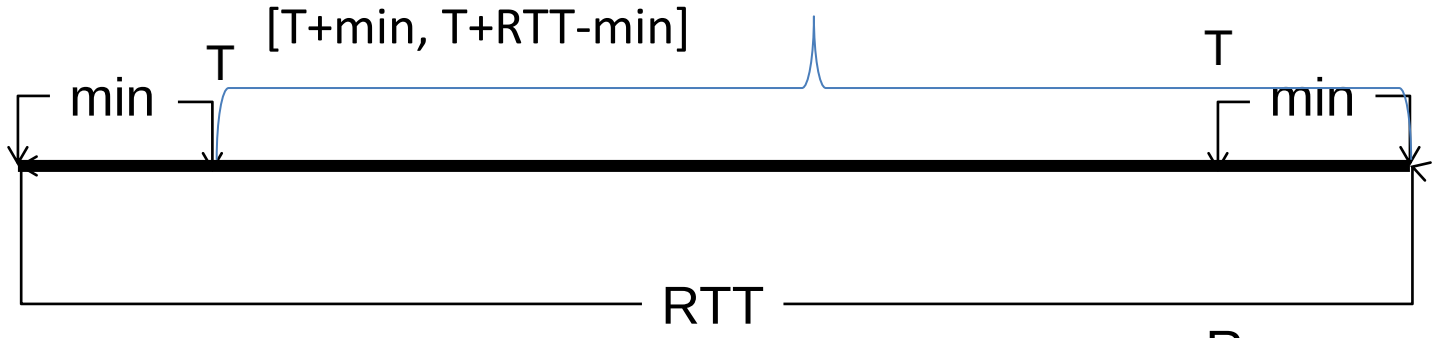
- Χρησιμοποιεί time server για τον συγχρονισμό
- Σχεδιασμένος για LAN
- Ο time server κρατά το χρόνο αναφοράς (π.χ. UTC)
- Ο client ζητά την ώρα, ο server απαντά με τη δική του ώρα T, ο client χρησιμοποιεί το T για να σετάρει το ρολόι του
- Ο χρόνος μετάδοσης μετ' επιστροφής (Round Trip Time – RTT) του δικτύου εισάγει καθυστέρηση
- Τι κάνουμε;
  - Εκτίμηση για την καθυστέρηση μιας διαδρομής
  - Ο client μπορεί να θέσει το ρολόι του σε T+RTT/2

Αλγόριθμος του Cristian

- RTT = χρόνος παραλαβής απάντησης – χρόνος αποστολής αιτήματος (υπολογισμένο στον client)

- Υποθέτουμε ότι ξέρουμε
  - Τον ελάχιστο χρόνο μετάδοσης min από τον client στον server
  - Ότι ο server έβαλε χρονοσφραγίδα στο μήνυμα ακριβώς πριν το στείλει

- Τότε ο πραγματικός χρόνος είναι ανάμεσα στις τιμές



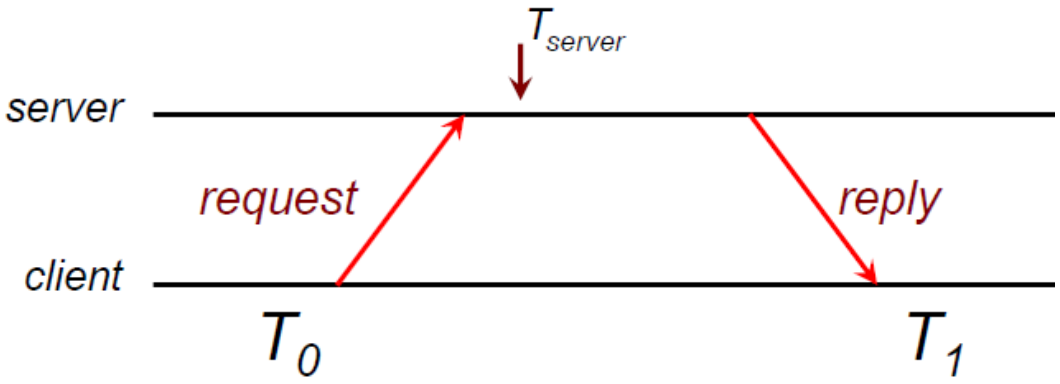
Αλγόριθμος του Cristian

- Η ακρίβεια είναι +-(RTT/2-min)

- Τελικά ο αλγόριθμος έχει ως εξής:
  - Ο client ρωτά τον time server
  - Ο time server στέλνει την ώρα T
  - Ο client βάζει το ρολόι του στην ώρα T+RTT/2

- Πώς μπορούμε να βελτιώσουμε την ακρίβεια;
  - Μέτρηση του RTT πολλές φορές για καλύτερη εκτίμηση του ελάχιστου -> μειώνουμε το σφάλμα
  - Για ασυνήθιστα μεγάλα RTT, τα αγνοούμε και επαναλαμβάνουμε -> απαλείφουμε τους outliers

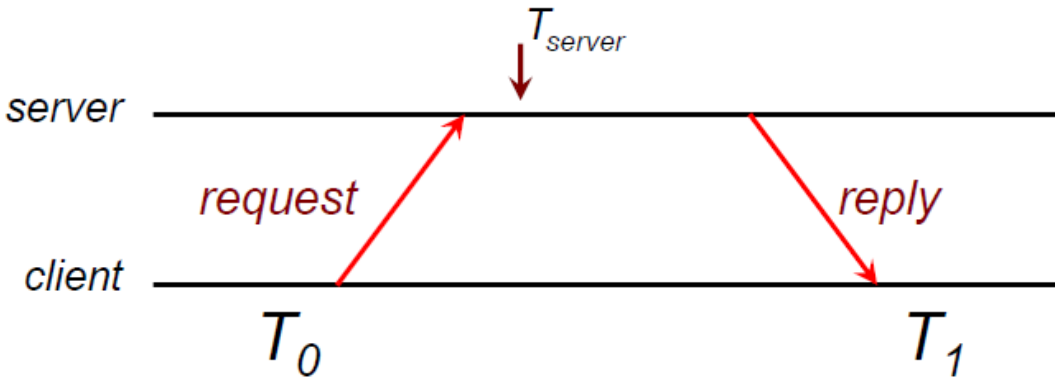
Παράδειγμα



- Αποστολή αιτήματος στις 5:08:15.100 (T0)
- Λήψη απάντησης στις 5:08:15.900 (T1)
- Η απάντηση είναι 5:09:25.300 (T)

- RTT = T1 -T0 = 5:08:15.900 - 5:08:15.100 = 800 ms
- Υποθέτουμε ότι ο χρόνος που πέρασε μέχρι να λάβουμε απάντηση είναι 400 ms
- Θέτουμε την ώρα T+ RTT/2
- 5:09:25.300 + 400 = 5:09.25.700

Παράδειγμα



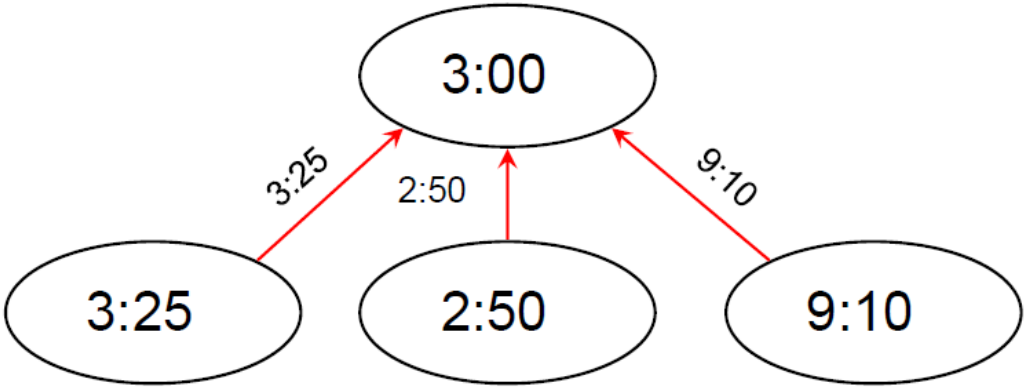
T_0 = 5:08:15.100
T_1 = 5:08:15.900
T_s = 5:09:25:300
T_{min} = 200 ms

- Αν γνωρίζαμε ότι min χρόνος είναι 200ms τότε
- Error = ± (900-100)/2-200 = ± 200 ms

Αλγόριθμος Berkeley

- Εσωτερικός συγχρονισμός
- 1 Master- οι υπόλοιποι slaves
- Ο Master ρωτάει την ώρα από τους slaves και υπολογίζει την τοπική τους ώρα με βάση το RTT (όπως στον αλγόριθμο του Cristian)
- Υπολογίζει τον μέσο όρο (συμπεριλαμβάνει και τη δική του ώρα)
- Στέλνει στον κάθε slave τη διαφορά (offset) από τον μέσο όρο (θετική ή αρνητική)
  - Γιατί offset και όχι ώρα;
- Δεν δουλεύει καλά όταν τα ρολόγια έχουν μεγάλες διαφορές
  - Απαλοιφή outliers

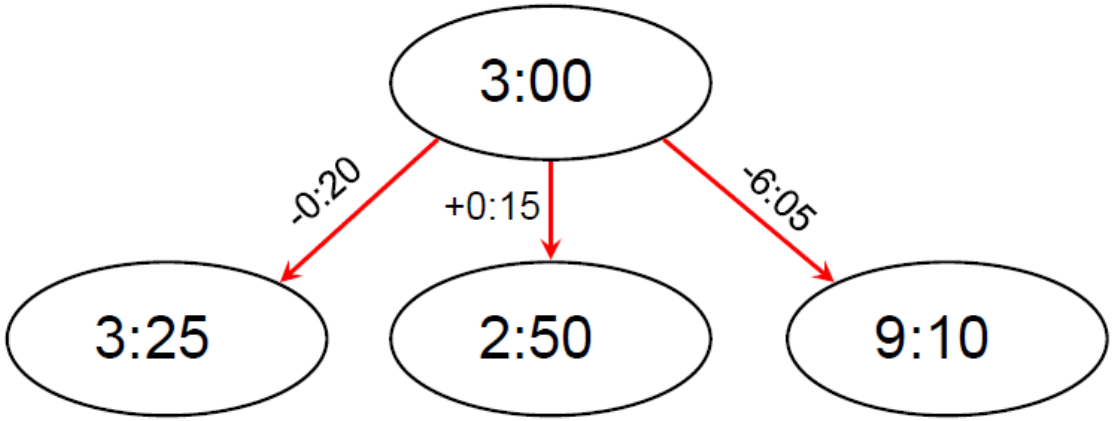
Παράδειγμα



1. Ζητάει ώρα από κάθε slave
2. Αγνοεί τους outliers (9:10)
3. Υπολογίζει μέσο όρο

$$\frac{3:25+2:50+3:00}{3}=3:05$$

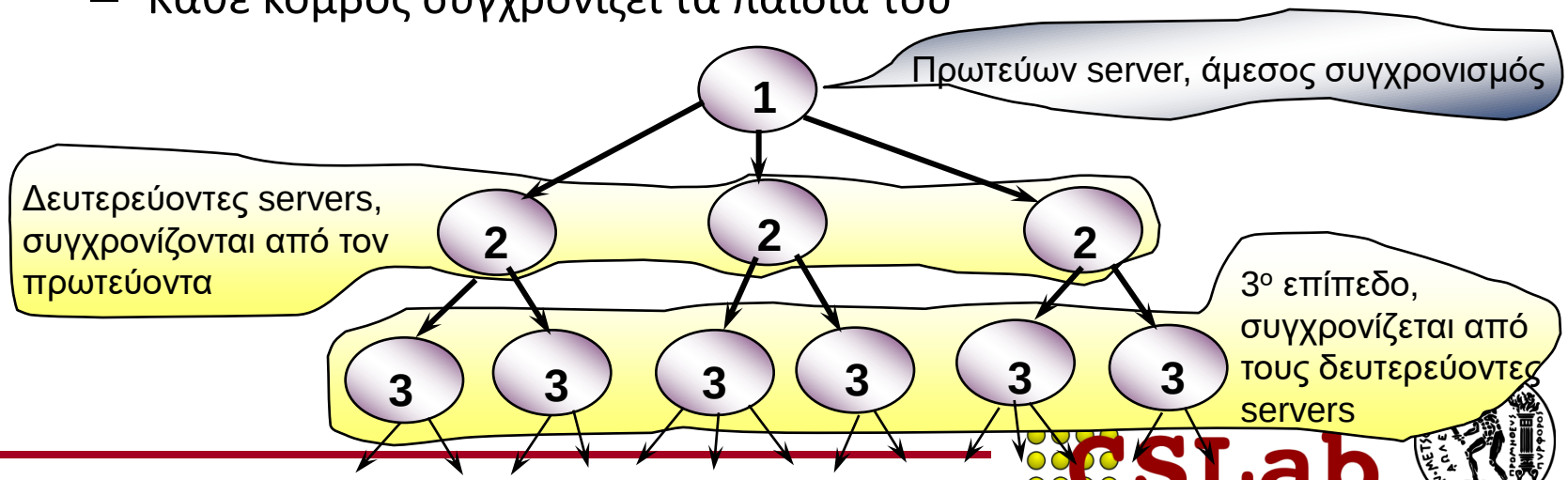
Παράδειγμα



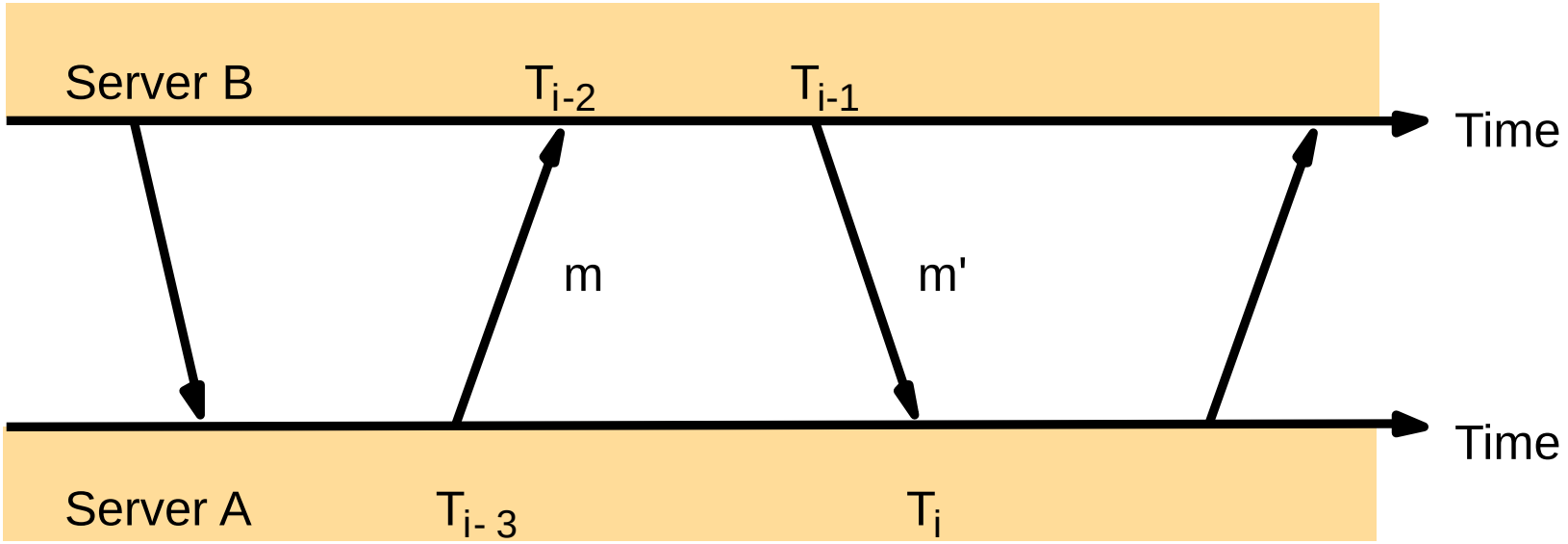
4. Στέλνει offset σε κάθε slave

Το πρωτόκολλο NTP

- Χρησιμοποιεί δίκτυο από time servers για τον συγχρονισμό
- ‘Εχει σχεδιαστεί για το διαδίκτυο
  - Γιατι όχι ο Cristian;
- Οι time servers είναι συνδεδεμένοι σε ένα δέντρο συγχρονισμού
  - Η ρίζα είναι σε επαφή με το UTC
  - Κάθε κόμβος συγχρονίζει τα παιδιά του

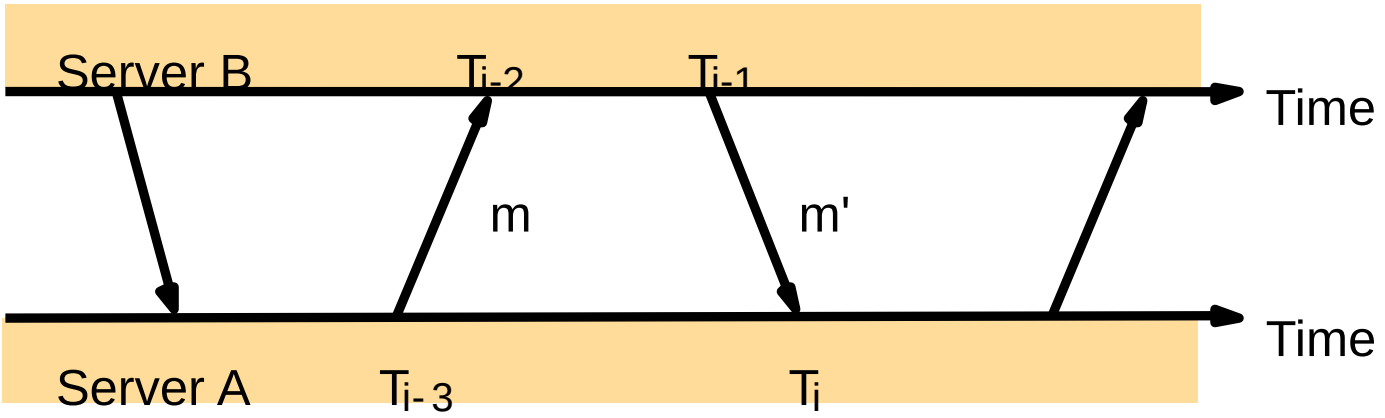


Ανταλλαγή μηνυμάτων



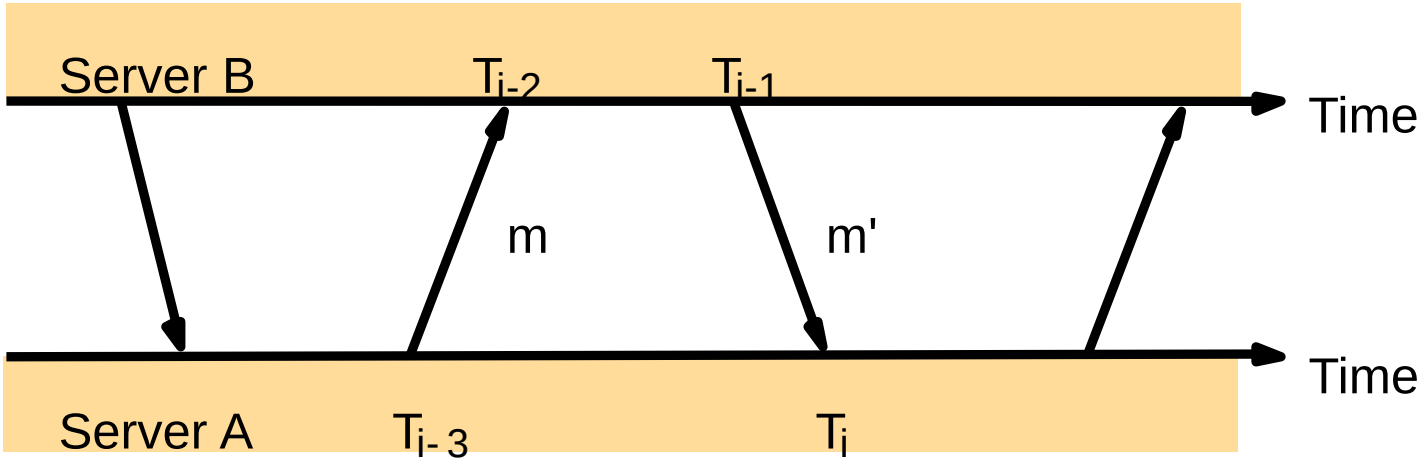
- Κάθε μήνυμα έχει χρονοσφραγίδες πρόσφατων γεγονότων:
  - Τοπική ώρα αποστολής και παραλαβής προηγούμενου NTP μηνύματος
  - Τοπική ώρα αποστολής τρέχοντος μηνύματος

Λίγη θεωρία



- o_i: εκτίμηση του offset ανάμεσα σε 2 ρολόγια
- d_i: η εκτίμηση για την καθυστέρηση: Συνολικός χρόνος αποστολής για *m* και *m'*
- Μια εκτίμηση για την καθυστέρηση είναι το ½ της συνολικής καθυστέρησης μείον το χρόνο επεξεργασίας στον απομακρυσμένο server

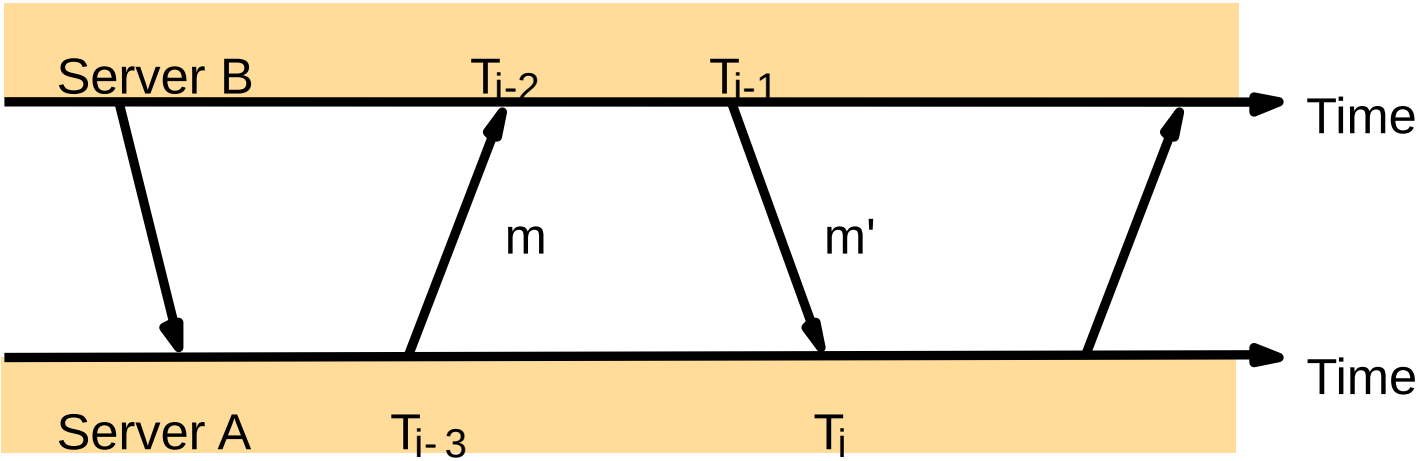
Το πρωτόκολλο (1)



- Υπολογισμός RTT: $(T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$
- Υπολογισμός καθυστέρησης για μια διαδρομή:

$$((T_i - T_{i-3}) - (T_{i-1} - T_{i-2}))/2$$

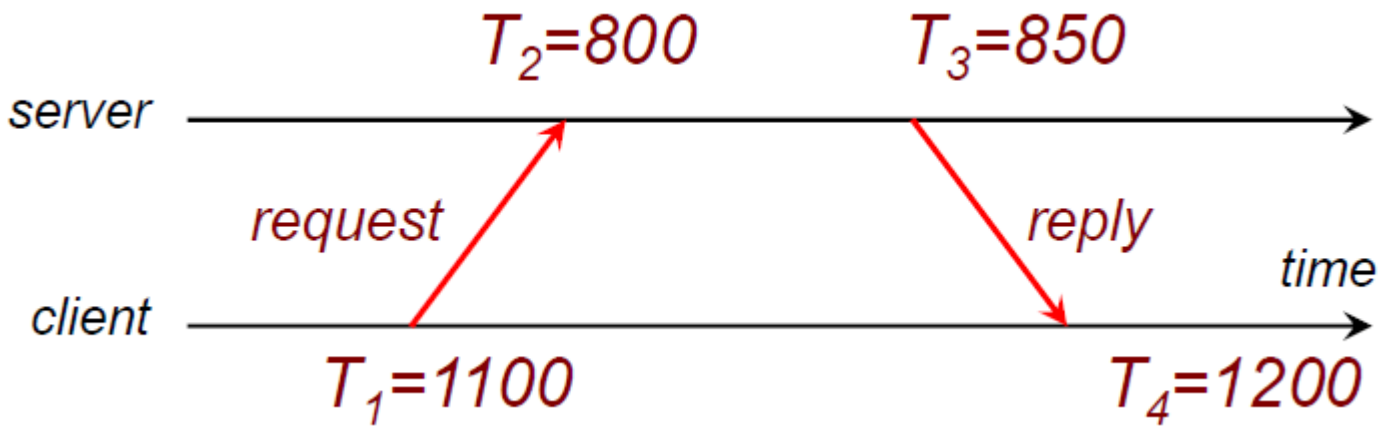
Το πρωτόκολλο (2)



- Υπολογισμός offset:

  T_{i-1} + (εκτίμηση χρόνου για μια διαδρομή) - T_i = ((T_{i-2} – T_{i-3}) + (T_{i-1} – T_i))/2
- Το ίδιο για πολλαπλούς εξυπηρετητές
- Στατιστική ανάλυση, απαλοιφή outliers, εφαρμογή αλγορίθμου data filtering

Παράδειγμα



Offset =

((800 - 1100) + (850 - 1200)) / 2

= ((-300) + (-350)) / 2

= -650 / 2 = -325

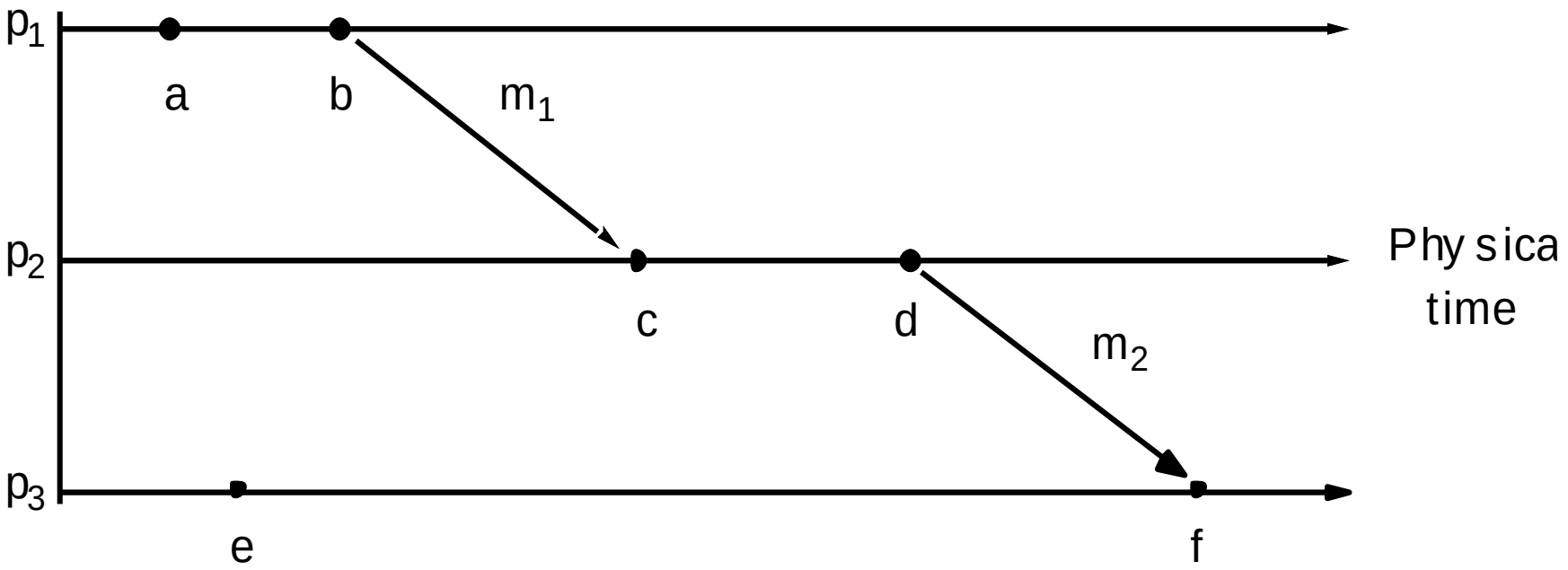
Set time to T_4 + o

= 1200 - 325 = 875

Επανάσταση

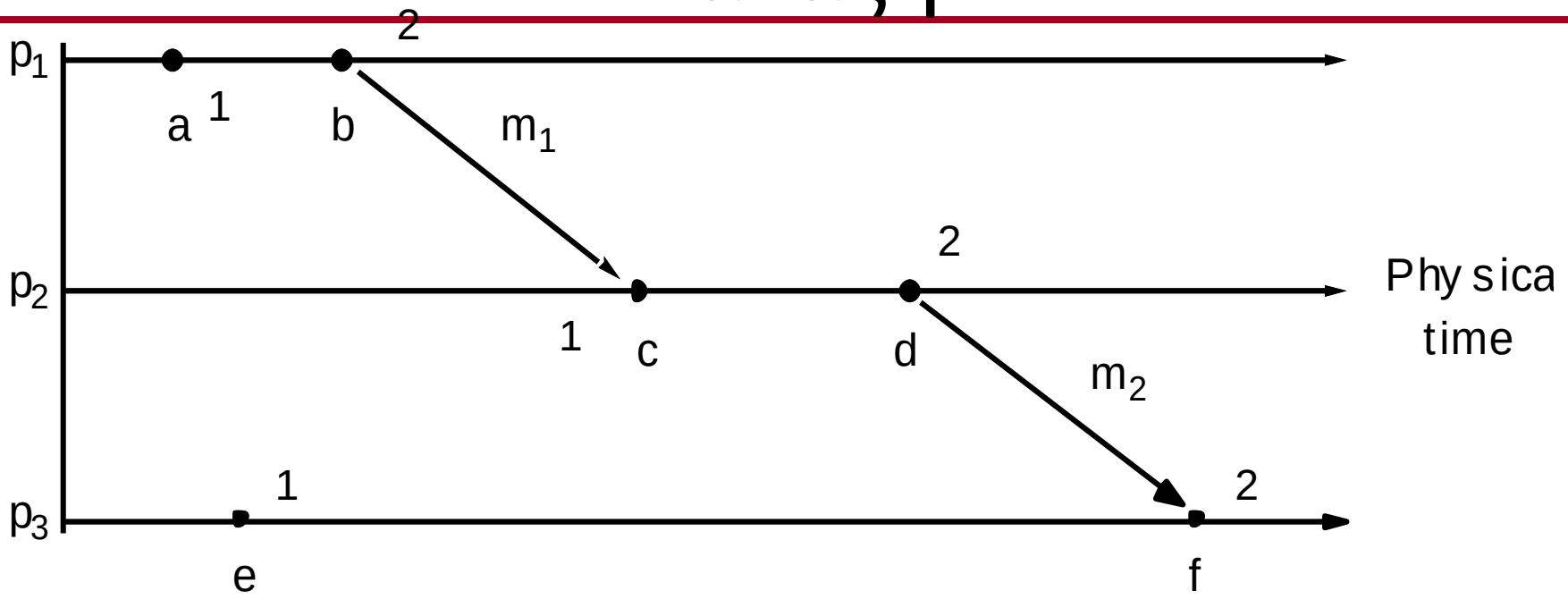
- Δεν μπορούμε να συγχρονίσουμε ρολόγια τέλεια
- Δεν μπορούμε να βασιστούμε σε φυσικά ρολόγια για διάταξη γεγονότων σε κατανεμημένες διεργασίες
- Λογικά ρολόγια
  - Το πρώτο προτάθηκε από τον Leslie Lamport το ‘70
  - Βασίζεται στην αιτιότητα (causality) των γεγονότων
  - Ορίζει σχετικό και όχι απόλυτο χρόνο
  - Βασική παρατήρηση: η χρονική διάταξη έχει νόημα μόνο όταν 2 ή περισσότερες διεργασίες επικοινωνούν με μηνύματα.

Το πρόβλημα



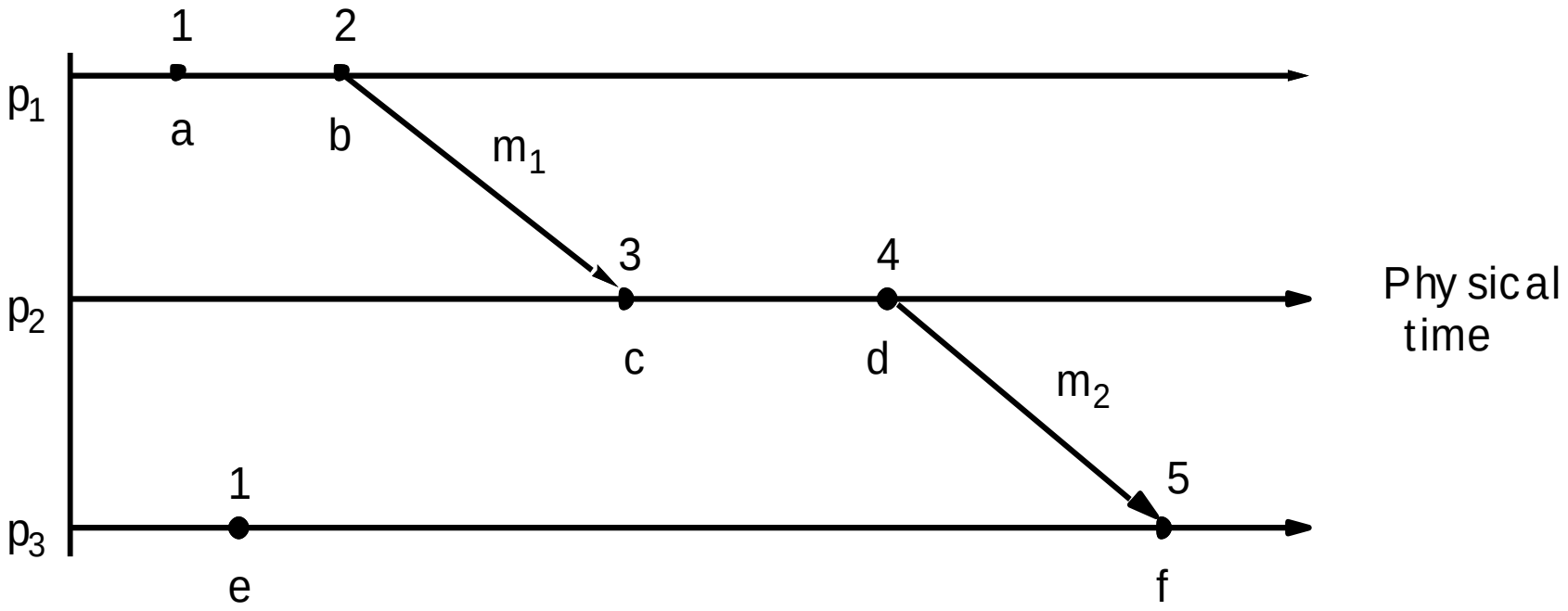
- Τι θέλουμε τελικά;
  - Να γνωρίζουμε για δύο events ποιο έγινε πριν το άλλο

Διάταξη



- Ιδανικά
  - Τέλειος συγχρονισμός φυσικών ρολογιών
- Αξιόπιστα
  - Events στην ίδια διεργασία p
  - Events αποστολής/λήψης μηνυμάτων

Χρονοσφραγίδες Lamport

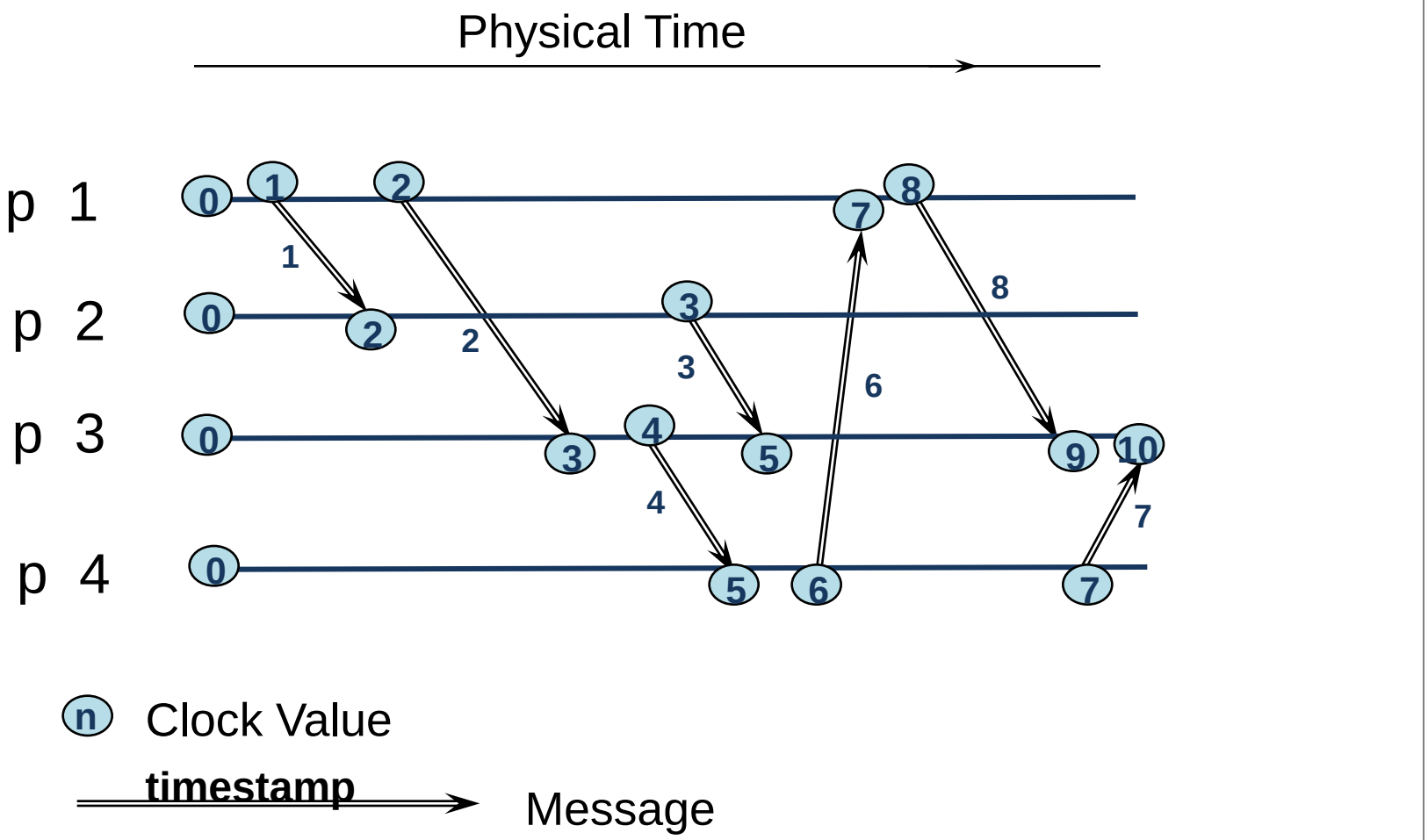


Λογικά Ρολόγια Lamport

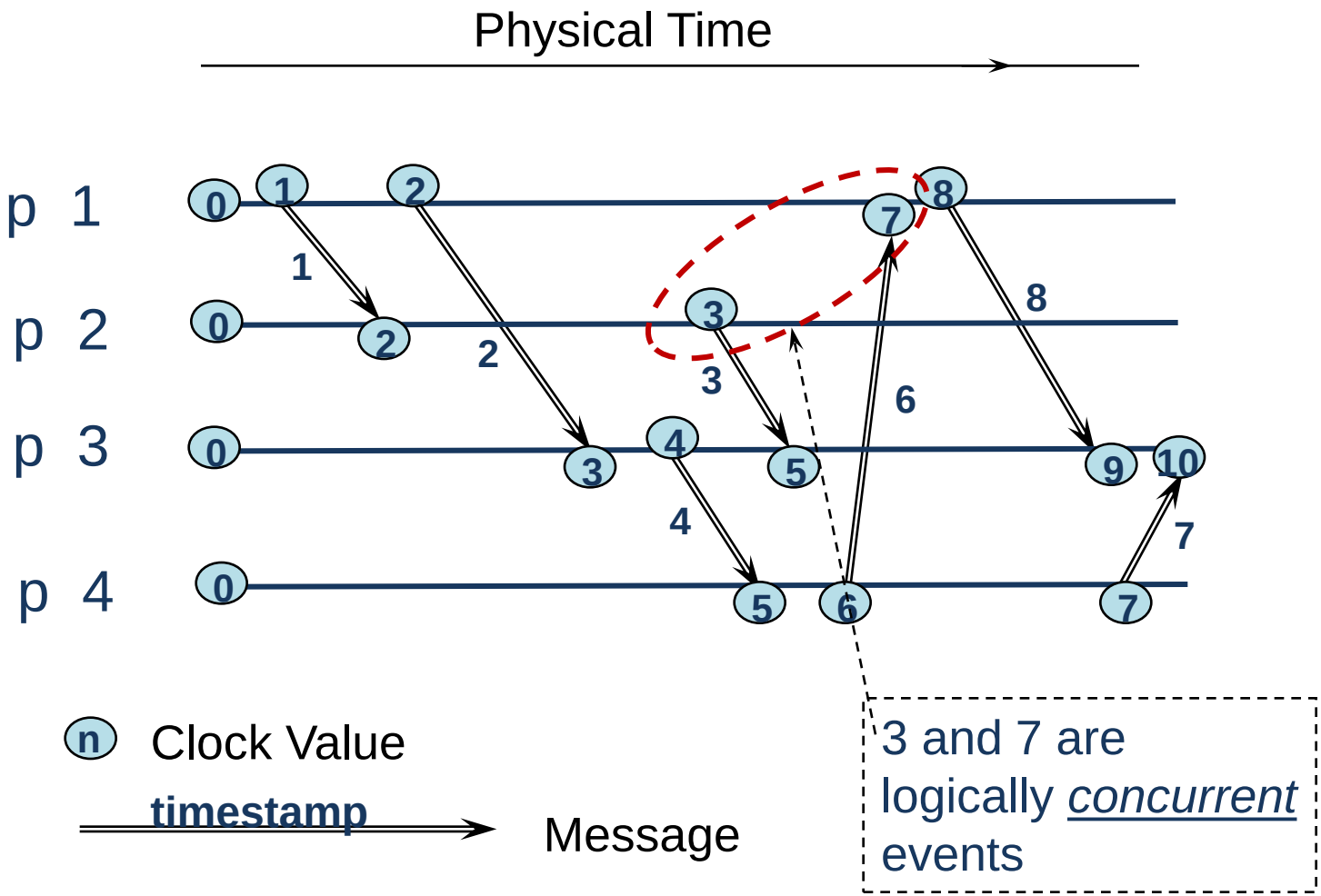
- Ο αλγόριθμος του Lamport αναθέτει *λογικές* χρονοσφραγίδες:
  - Όλες οι διεργασίες έχουν έναν μετρητή (ρολόι) με αρχική τιμή 0
  - Μια διεργασία αυξάνει τον μετρητή της όταν συμβαίνει ένα event αποστολής ή μια λειτουργία. Η τιμή του μετρητή δίνεται ως χρονοσφραγίδα του event
  - Ένα μήνυμα φέρει τη χρονοσφραγίδα του event αποστολής
  - Για κάθε λήψη μηνύματος η διεργασία ανανεώνει τον μετρητή της δίνοντας τιμή max(local clock, message timestamp) + 1

- Ορισμός της λογικής σχέσης *happened-before* ($\rightarrow$) ανάμεσα σε events:
  - Για γεγονότα εντός μιας διεργασίας: $a \rightarrow b$, αν $time(a) < time(b)$
  - Αν η p1 στείλει *m* στο p2: $send(m) \rightarrow receive(m)$
  - (Μεταβατικότητα) Αν $a \rightarrow b$ και $b \rightarrow c$ τότε $a \rightarrow c$
  - Δείχνει αιτιότητα γεγονότων

Παράδειγμα


Physical Time
p 1
p 2
p 3
p 4
0
1
2
7
8
1
0
2
2
3
8
3
6
0
3
4
5
9
10
4
7
0
5
6
7
n
Clock Value
timestamp
Message

Ένα θεματάκι



Ολική διάταξη

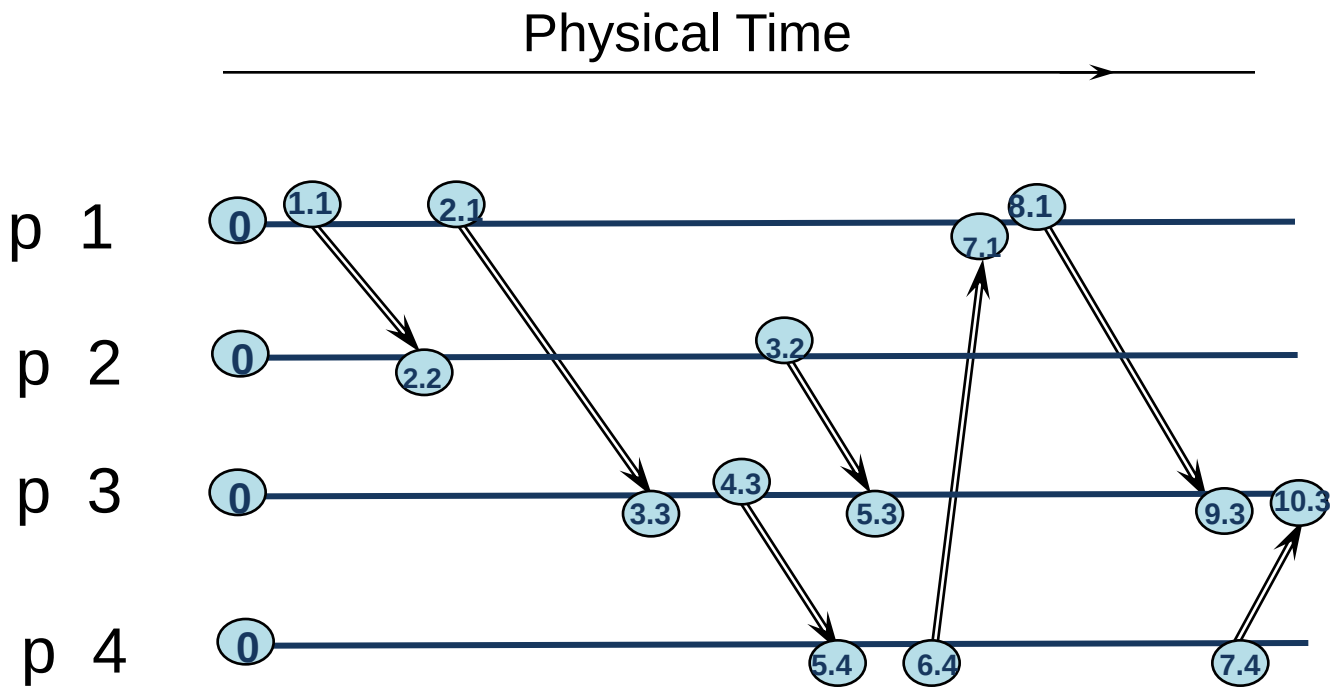
- Λογικά ταυτόχρονα events μπορεί να έχουν ίδιες χρονοσφραγίδες… ή και όχι
- Μπορούμε να επιβάλλουμε καθολικά μοναδικές χρονοσφραγίδες (T_i, i)
  - Lamport timestamp της διεργασίας i
  - Μοναδικό id της διεργασίας I (π.χ. Host address, process id)

- Σύγκριση χρονοσφραγίδων
  - (T_i, i) < (T_j, j) όταν

    $T_i < T_j$ ή

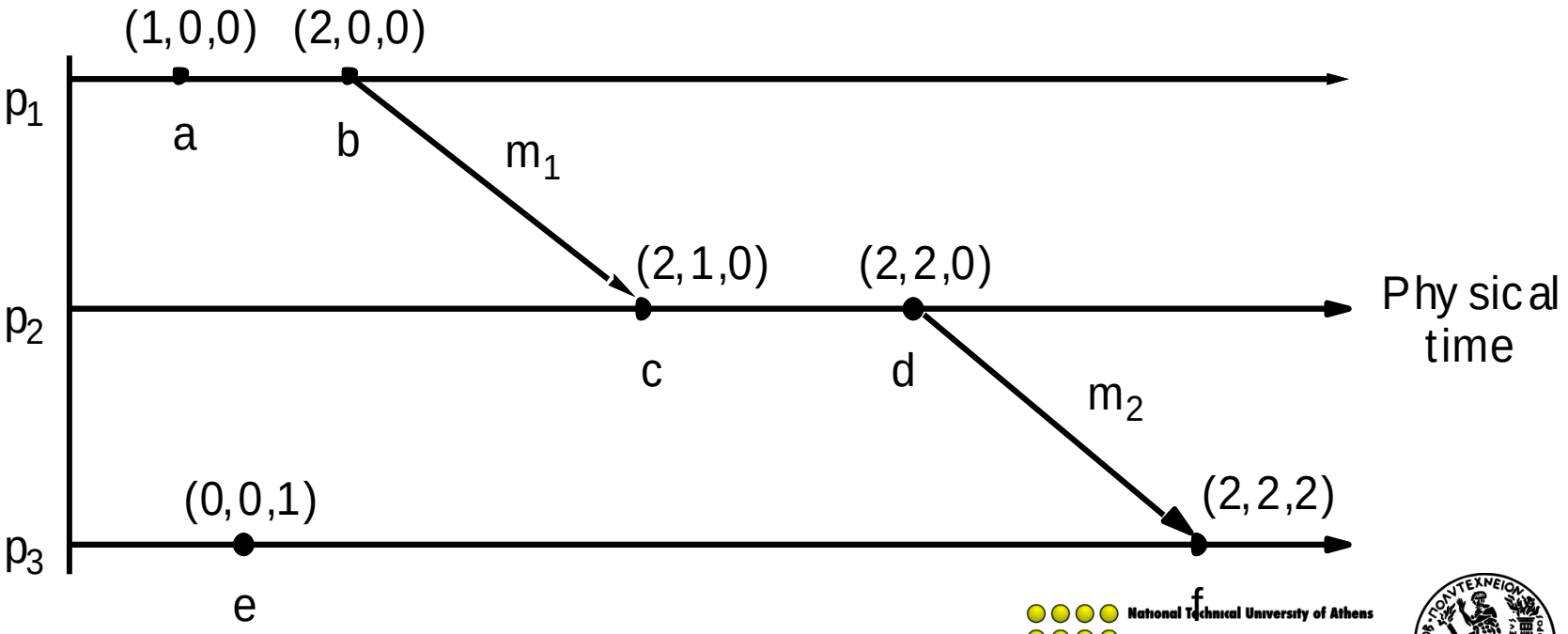
    $T_i = T_j$ και i<j
- Δεν αντιστοιχεί απαραίτητα σε πραγματική διάταξη

Παράδειγμα


Physical Time
p 1
p 2
p 3
p 4
0
1.1
2.1
8.1
7.1
0
2.2
3.2
0
3.3
4.3
5.3
9.3
10.3
0
5.4
6.4
7.4
n
Clock Value

Διανυσματικές χρονοσφραγίδες

- Με τα ρολόγια Lamport
  - e “happened-before” f $\Rightarrow$ timestamp(e) < timestamp (f), αλλά
  - timestamp(e) < timestamp (f) $\not\Rightarrow$ e “happened-before” f



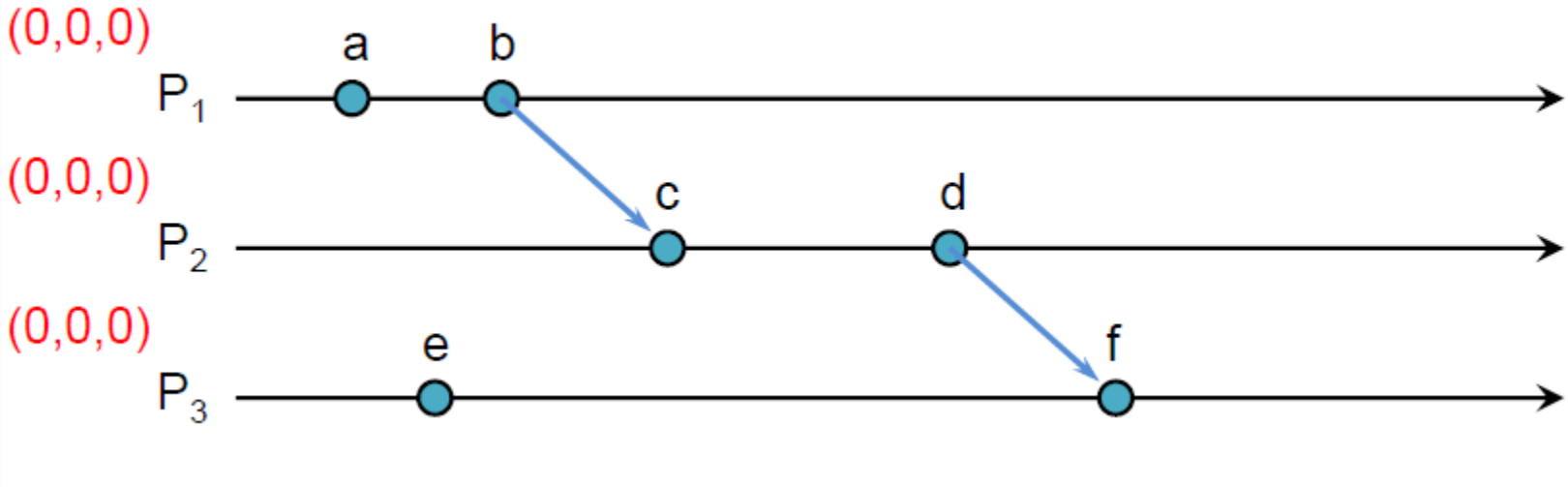
Λογικά διανυσματικά ρολόγια

- Ο διανυσματικός λογικός χρόνος:
  - Όλες οι διεργασίες χρησιμοποιούν διανύσματα μετρητών (λογικά ρολόγια), το i στοιχείο του ρολογιού είναι ο μετρητής της διεργασίας i, αρχικά όλα 0
  - Κάθε διεργασία i αυξάνει το στοιχείο i του διανύσματος σε κάθε event λειτουργίας ή αποστολής. Η τιμή του διανύσματος είναι το timestamp του event
  - Ένα μήνυμα αποστέλλεται με το vector timestamp
  - Για ένα event παραλαβής μηνύματος
    - $V_{receiver}[j] = Max(V_{receiver}[j] , t[j]),$ για j=1, 2, ..., N
    - $V_{receiver}[j] + 1$, j=receiver

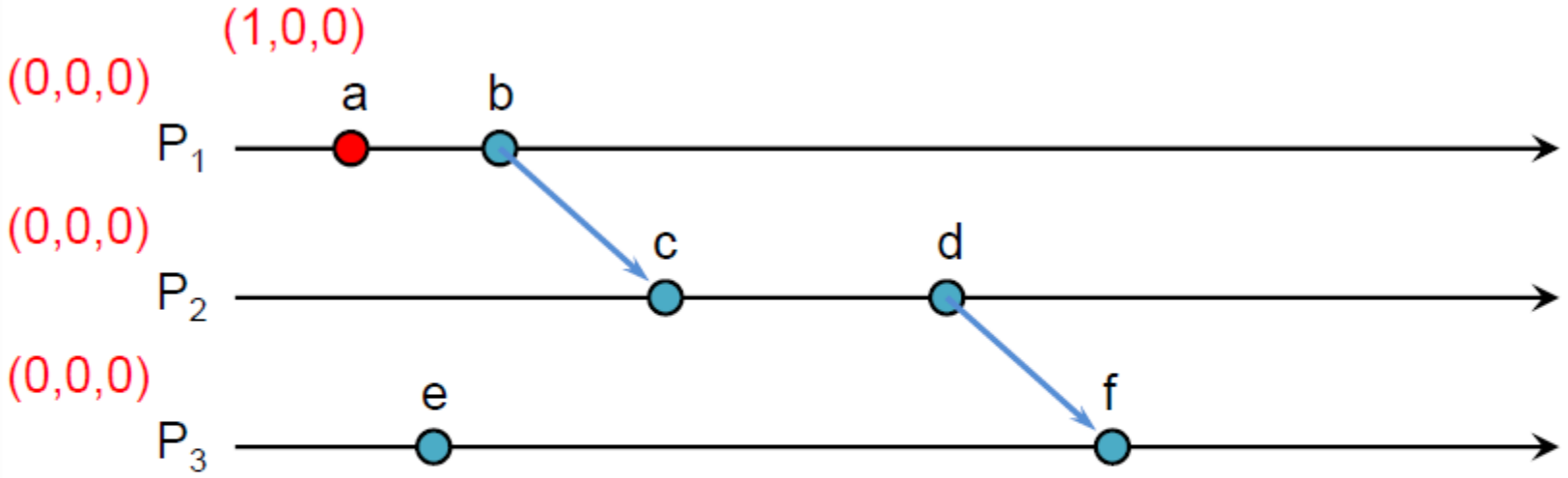
Σύγκριση διανυσματικών χρονοσφραγίδων

- $VT_1 = VT_2$,
  - *iff* $VT_1[i] = VT_2[i]$, for all i = 1, … , n
- $VT_1 <= VT_2$,
  - *iff* $VT_1[i] <= VT_2[i]$, for all i = 1, … , n
- $VT_1 < VT_2$,
  - *iff* $VT_1 <= VT_2$ & ∃ j (1 <= j <= n & $VT_1[j] < VT_2[j]$)
- VT_1 is concurrent with VT_2
  - *iff* (not $VT_1 <= VT_2$ AND not $VT_2 <= VT_1$)

Παράδειγμα 1

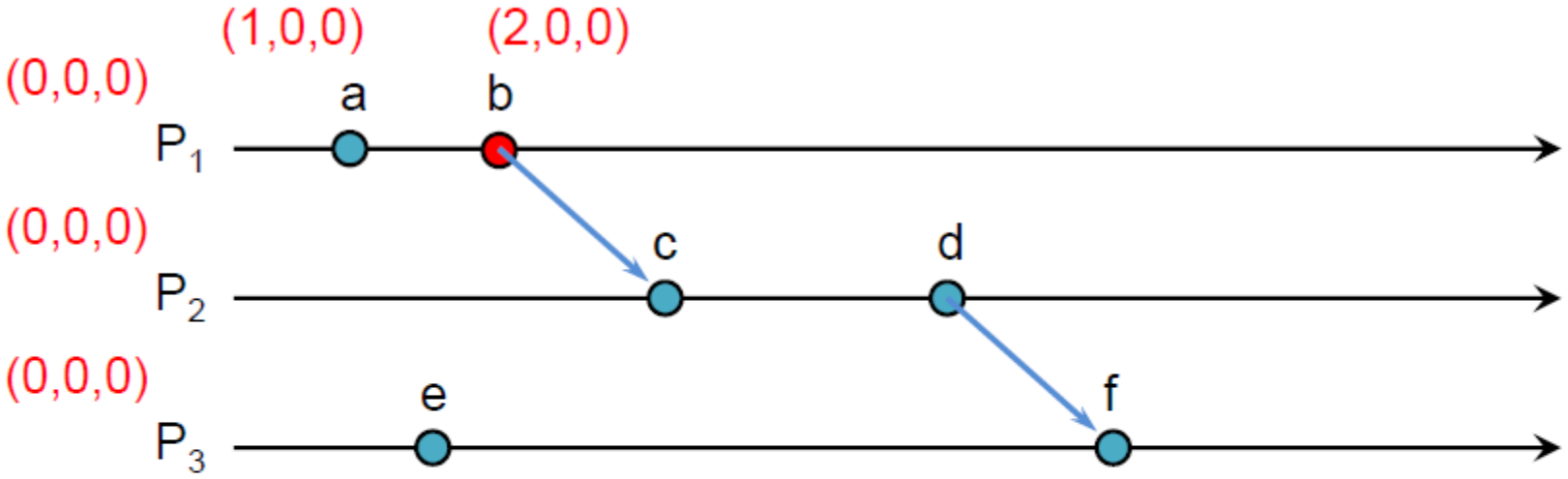


Παράδειγμα 1



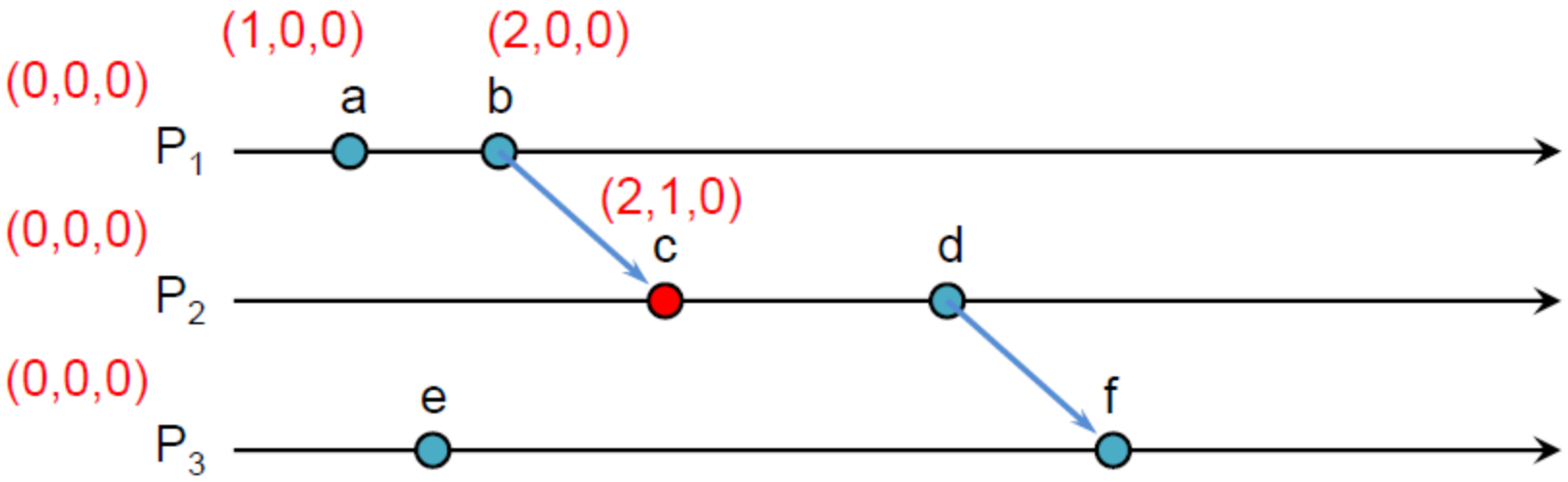
Event	timestamp
a	(1,0,0)

Παράδειγμα 1



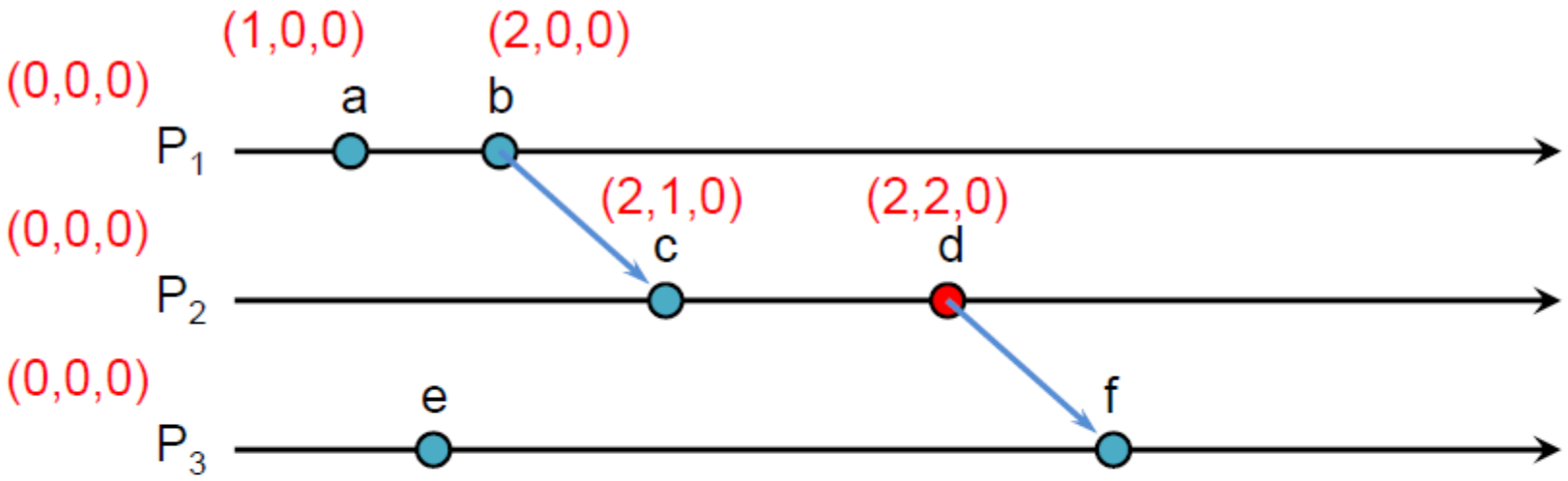
Event	timestamp
a	(1,0,0)
b	(2,0,0)

Παράδειγμα 1



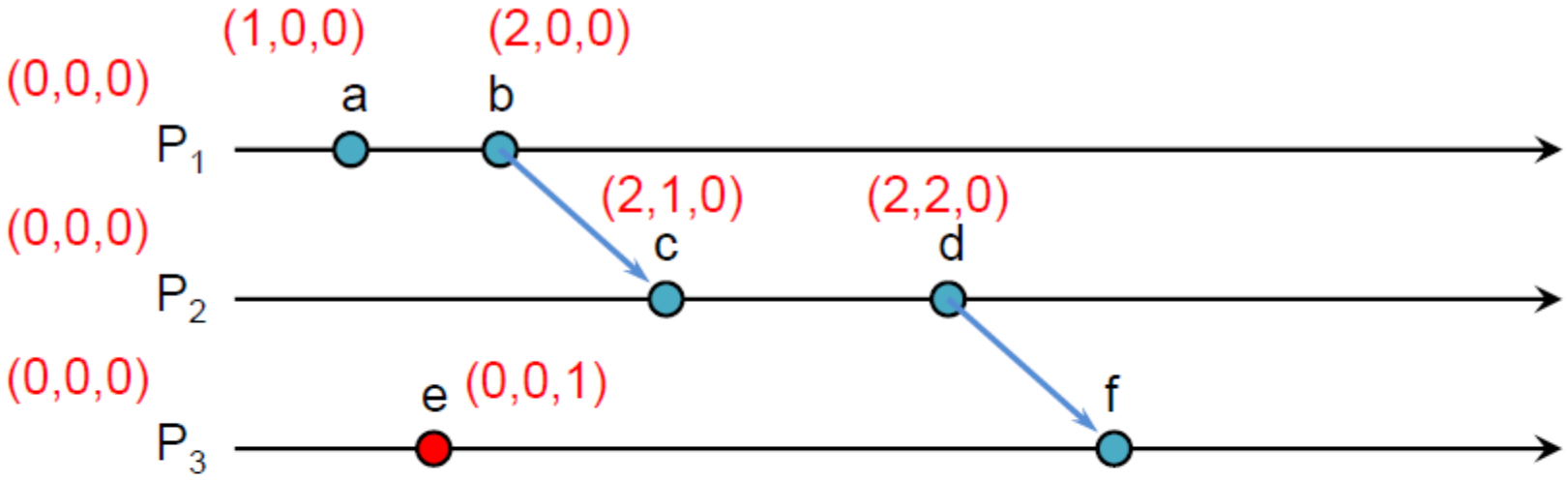
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

Παράδειγμα 1



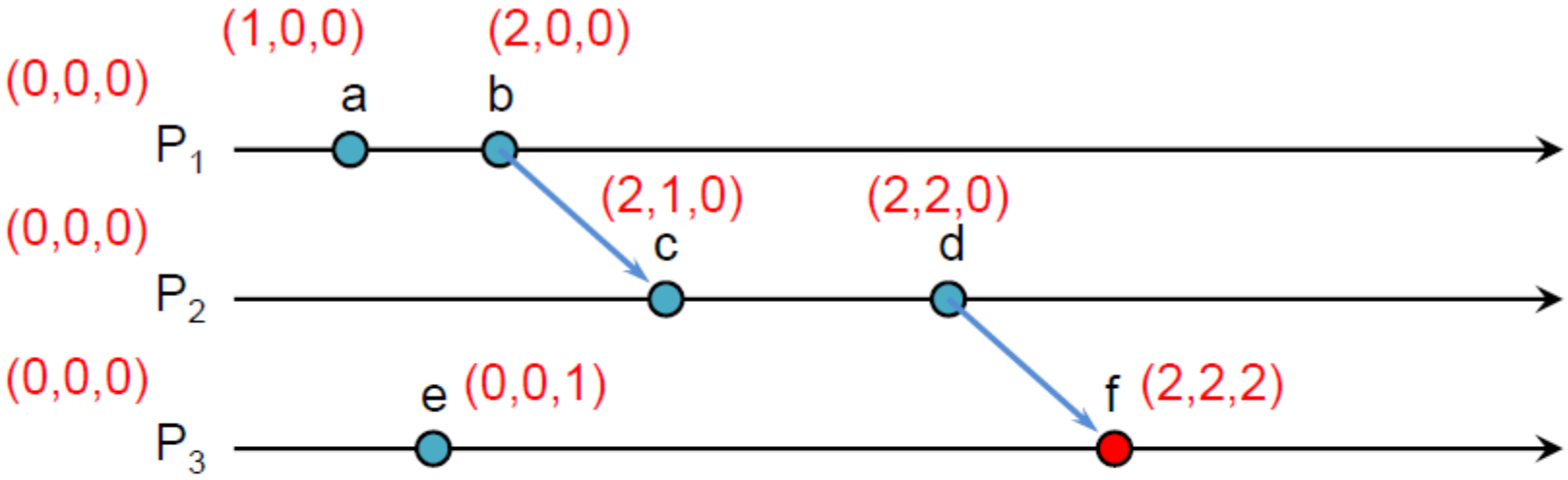
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

Παράδειγμα 1



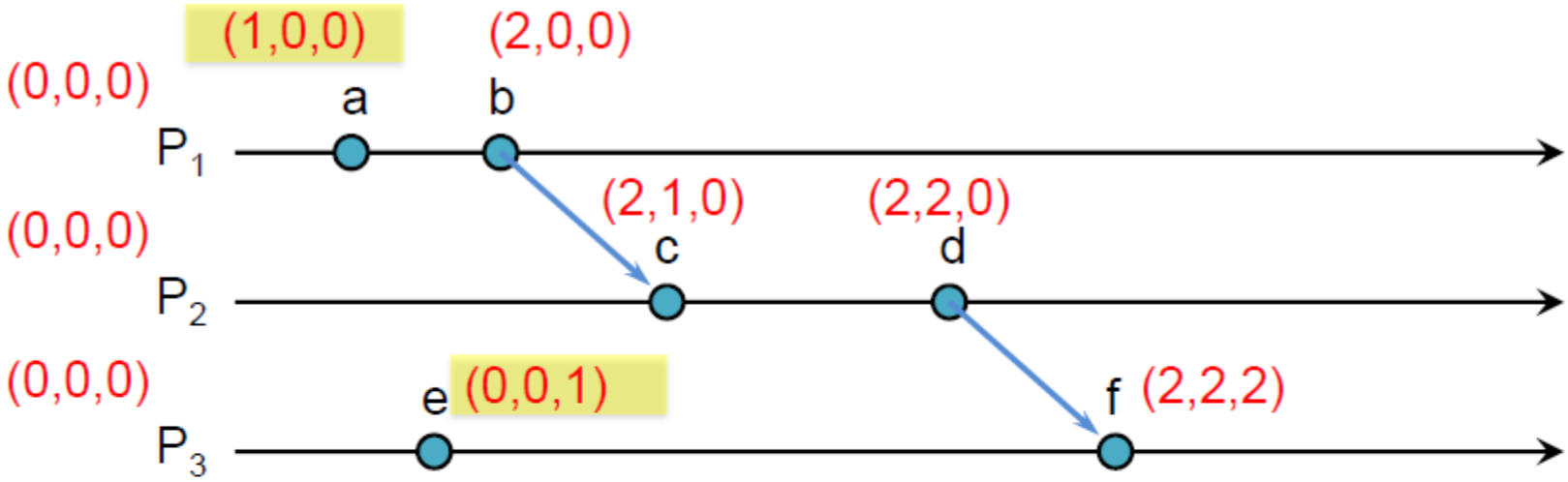
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)

Παράδειγμα 1



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

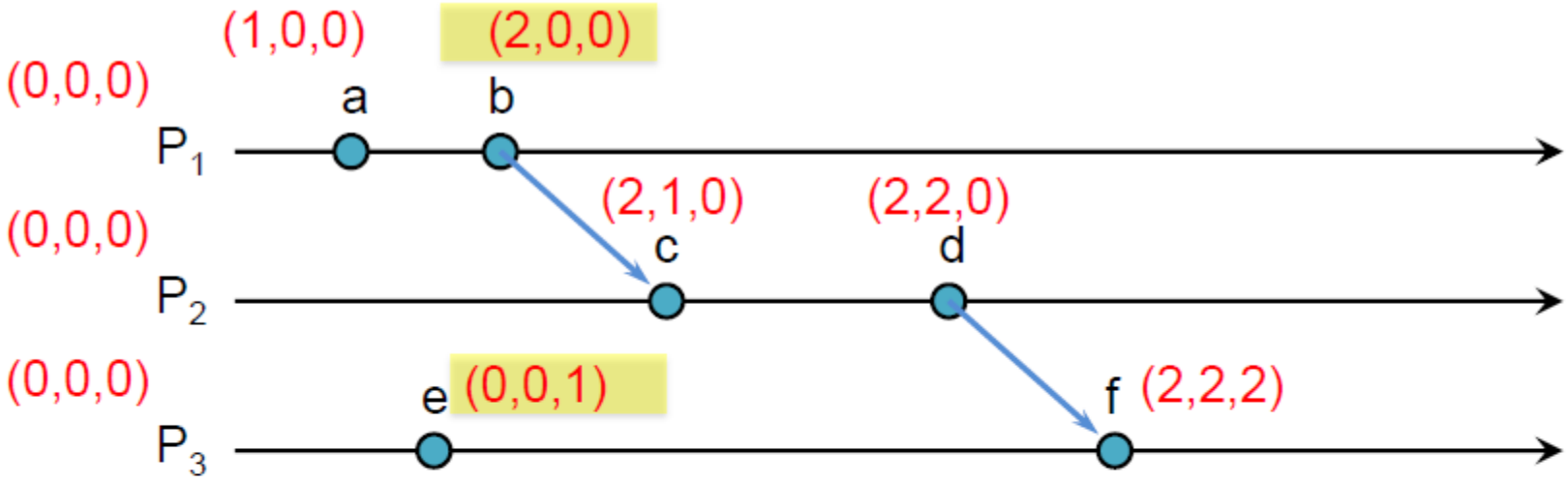
Παράδειγμα 1



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

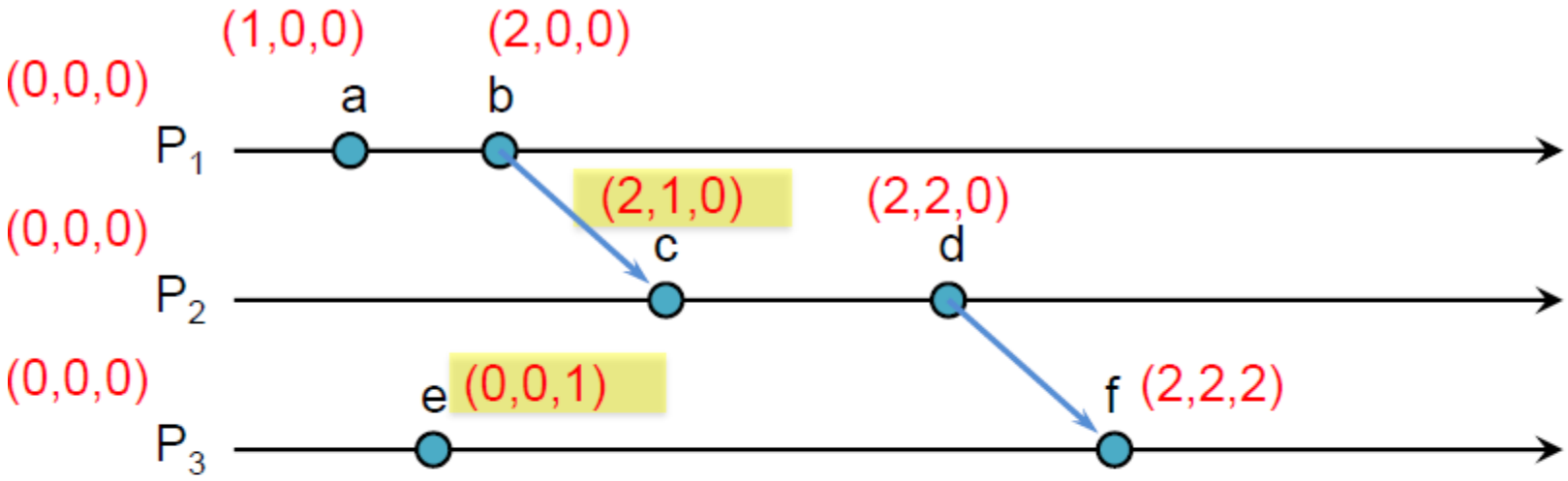
Παράδειγμα 1



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

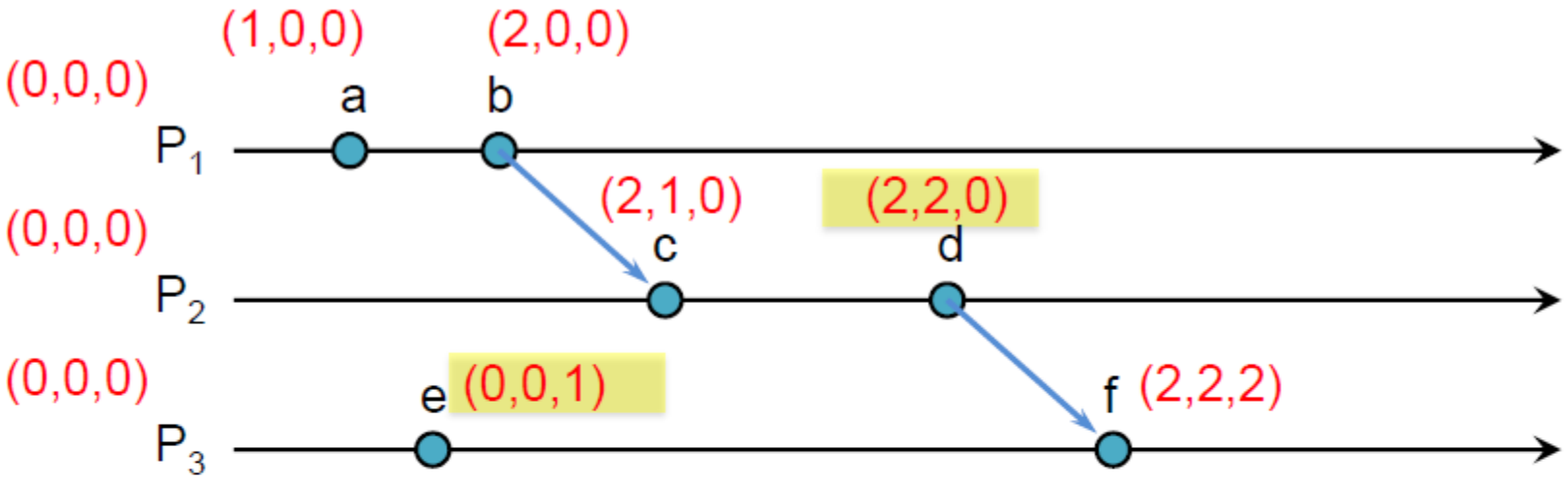
Παράδειγμα 1



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

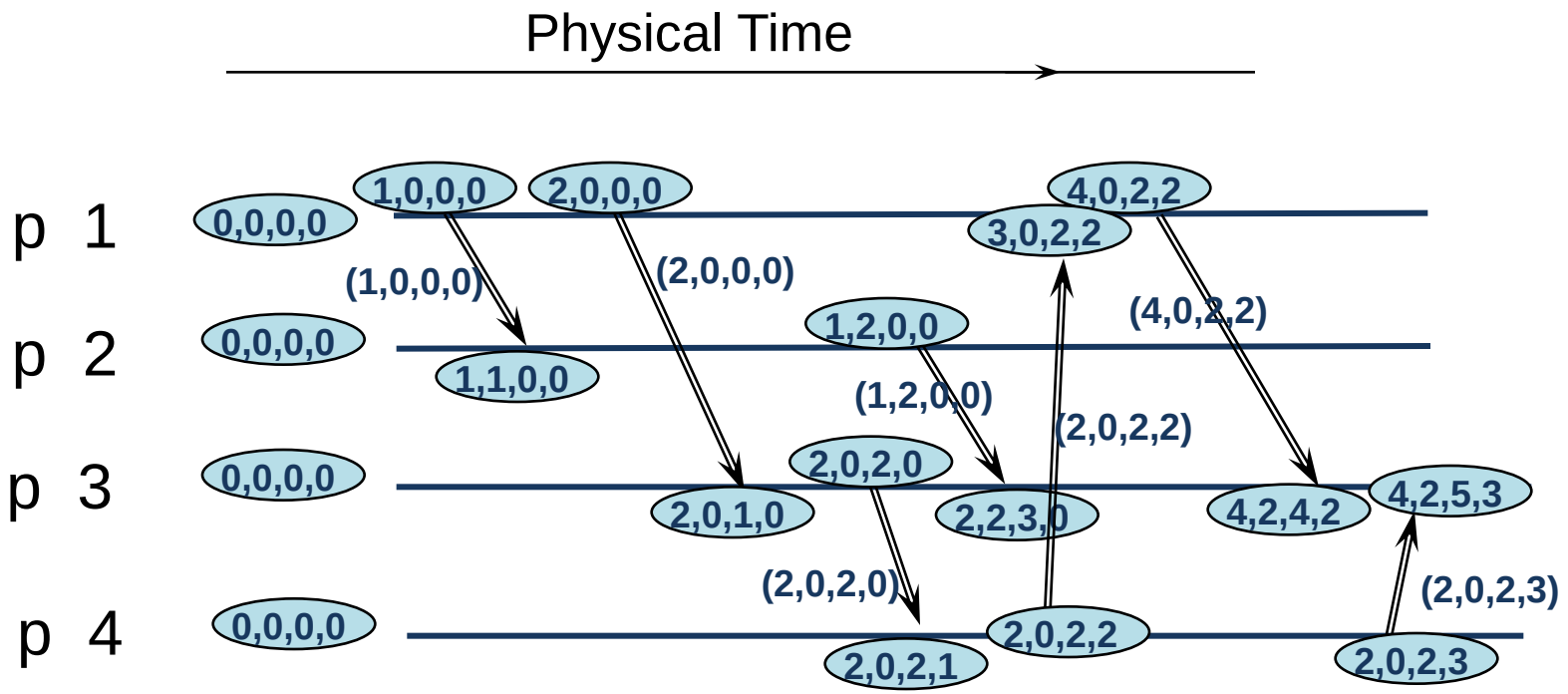
Παράδειγμα 1



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

Παράδειγμα 2



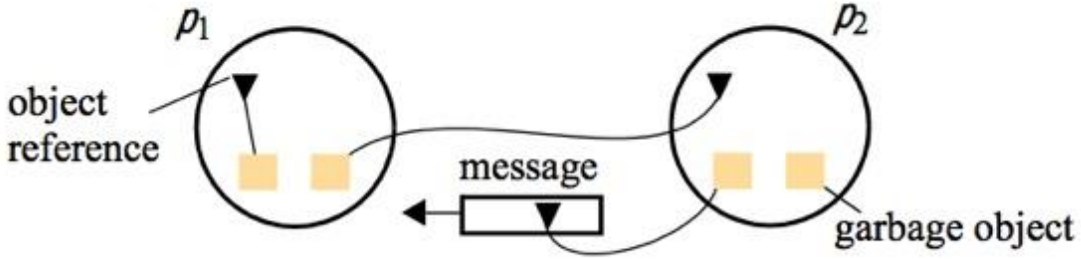
Διανυσματικά ρολόγια

- Με τα διανυσματικά ρολόγια
  - e “happened-before” f $\Rightarrow$ V(e) < V(f) και
  - V(e) < V(f) $\Rightarrow$ e “happened-before” f
- Έχει αποδειχθεί ότι οι N διαστάσεις είναι αναπόφευκτες
- Μειονέκτημα σε σχέση με τα ρολόγια Lamport;;

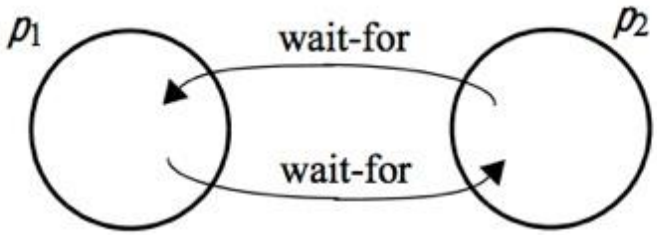
Η χρήση λογικών ρολογιών

- Είναι απόφαση σχεδιασμού
- NTP error bound
  - Τοπικά: λίγα ms
  - Wide-area: δεκάδες ms
- Αν το σύστημα δεν επηρεάζεται από αυτήν την ανακρίβεια -> NTP
- Τα λογικά ρολόγια επιβάλλουν τυχαία διάταξη σε ταυτόχρονα events
  - Breaking ties: process IDs, κλπ.

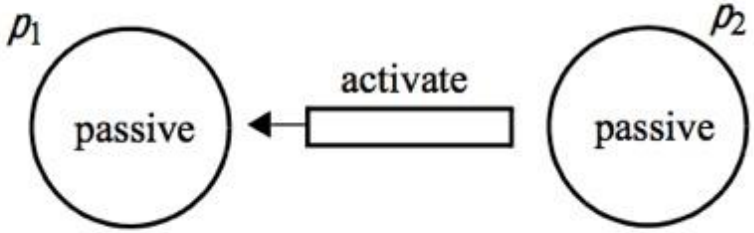
Καθολικές Καταστάσεις



(a) Garbage collection



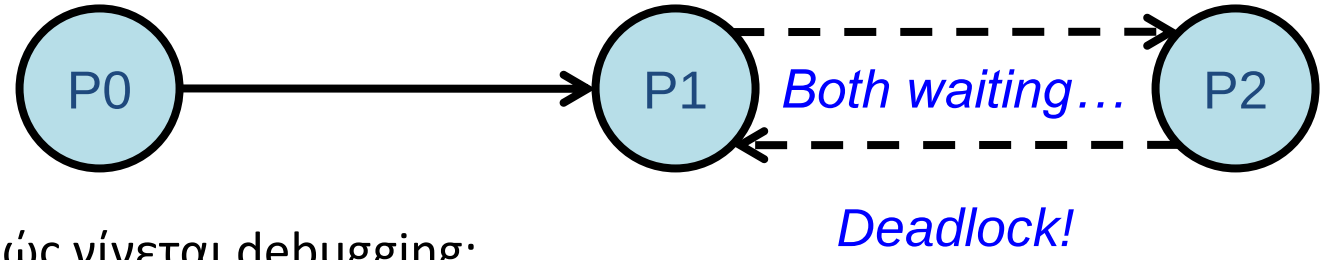
(b) Deadlock



(c) Termination

Παράδειγμα

- Distributed debugging

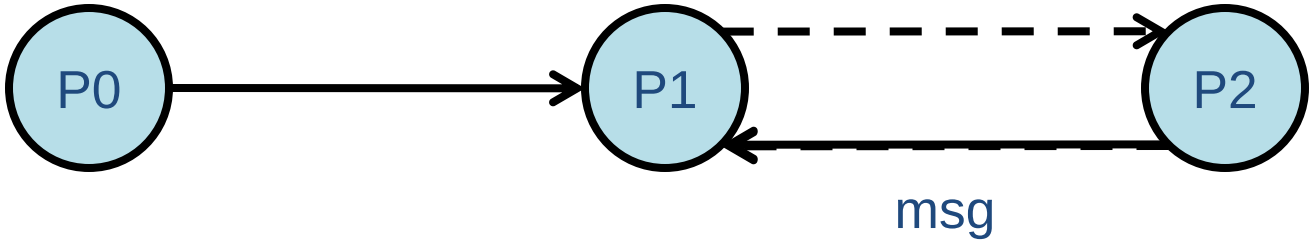


- Πώς γίνεται debugging;
  - Βρίσκοντας το ολικό snapshot!

- Θέλουμε 3 πράγματα
  - Κατάσταση ανά process
  - Μηνύματα στο δίκτυο
  - Όλα τα events που έγιναν πριν από κάθε event στο snapshot

Πρώτη απόπειρα

- Συγχρονίζουμε τα ρολόγια όλων των διεργασιών
  - Όλες οι διεργασίες καταγράφουν την κατάστασή τους τη στιγμή *t*
- Προβλήματα?
  - Ο συγχρονισμός των ρολογιών γίνεται μόνο κατά προσέγγιση
  - Δεν καταγράφονται τα μηνύματα στο δίκτυο



- Η διάταξη των γεγονότων είναι αρκετή
- Χρειαζόμαστε ένα λογικό ολικό snapshot
  - Κατάσταση κάθε διεργασίας
  - Μηνύματα καθ' οδόν σε όλα τα κανάλια επικοινωνίας

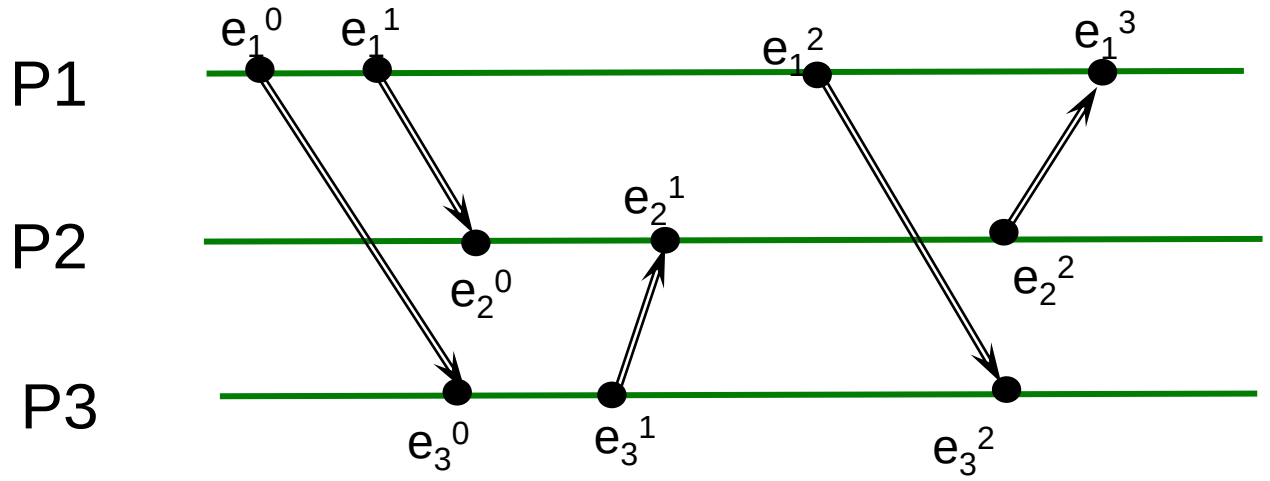
Πώς;

- Μπορούμε από τα local states των διεργασιών να εξάγουμε ένα global state που να έχει νόημα;

  Ναι!

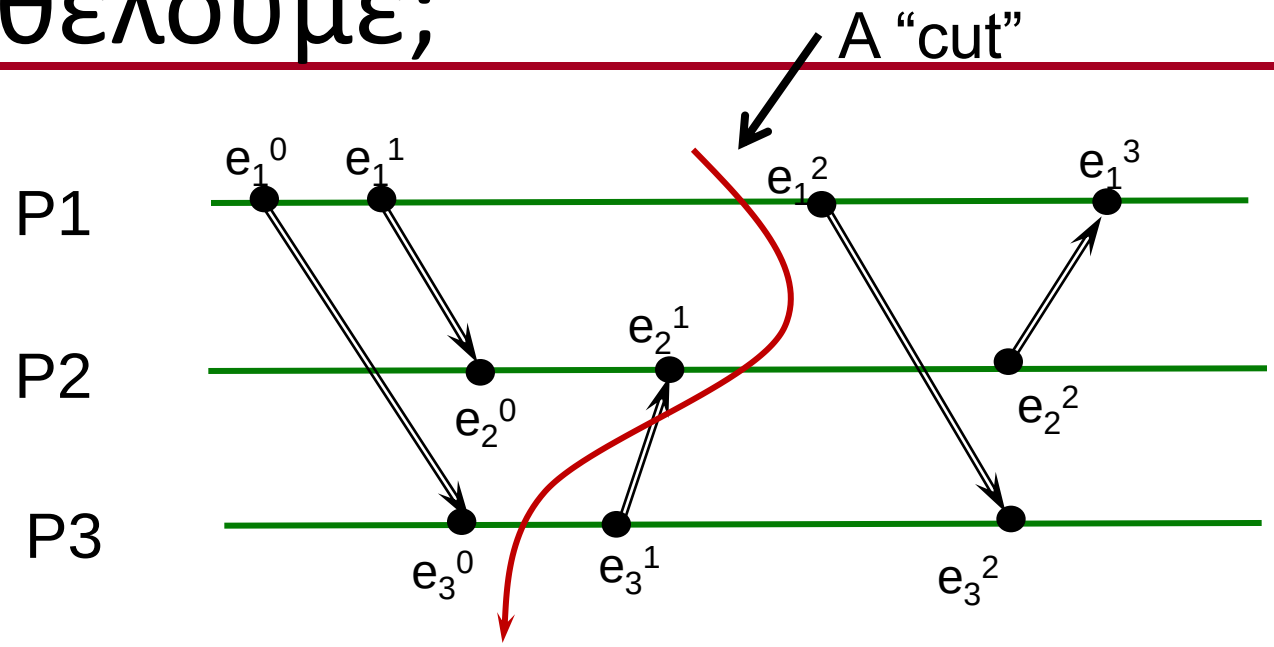
  Πρώτα όμως μερικοί ορισμοί…

Ορισμοί



- Για μια διεργασία P_i , όπου συμβαίνουν τα e_i^0, e_i^1, …,
  - *history*(P_i) = $h_i = <e_i^0, e_i^1, \ldots>$
  - *prefix history*(P_i^k) = $h_i^k = <e_i^0, e_i^1, \ldots, e_i^k>$
  - S_i^k : *η κατάσταση του* P_i αμέσως πριν το event k
- Για ένα σύνολο διεργασιών P_1 , …,P_i , …. :
  - Global history: $H = \cup_i (h_i)$
  - Global state: $S = \cup_i (S_i^{k_i})$
  - A cut $C \subseteq H = h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_n^{cn}$
  - The frontier of $C = \{e_i^{ci}, i = 1,2, \ldots n\}$

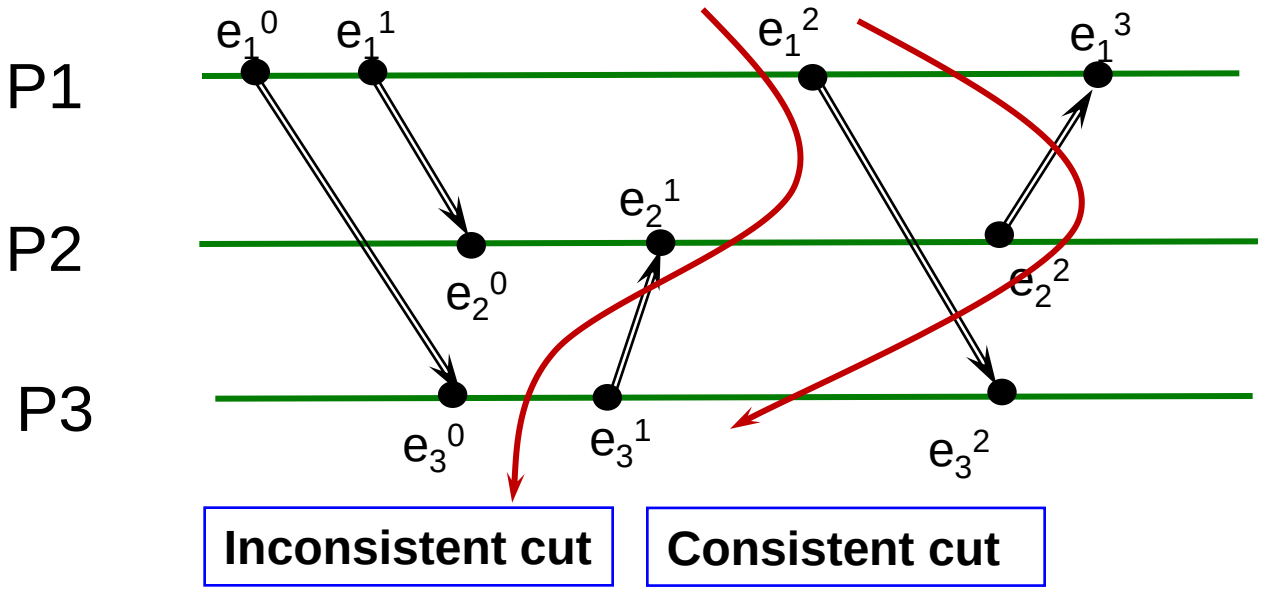
Τι θέλουμε;



- Είναι καλό αυτό το snapshot?
  - Όχι γιατί το e_2^1 προκλήθηκε από το e_3^1.

Συνεπείς Καταστάσεις

- Ένα cut C είναι συνεπές (consistent cut) αν
  - $\forall_{e \in C}$ *(αν* $f \rightarrow e$ *τότε* $f \in C$*)*
- Μια καθολική κατάσταση S είναι συνεπής όταν
  - Αντιστοιχεί σε consistent cut



Άρα

- Ένα consistent cut αντιστοιχεί σε ένα consistent global state.
  - Είναι ένα πιθανό state του συστήματος
  - Ίσως όμως η πραγματική εκτέλεση να μην πέρασε τελικά από αυτό το state

Γιατί συνεπείς καταστάσεις;

- #1: Για κάθε event, μπορείς να βρεις την αιτιότητα (trace back)
- #2: Μηχανή καταστάσεων (state machine)
  - Η εκτέλεση ενός κατανεμημένου αλγορίθμου είναι μια αλληλουχία από transitions ανάμεσα σε καθολικές καταστάσεις : S0 → S1 → S2 → …
  - …όπου κάθε transition γίνεται με μια μοναδική πράξη μιας διεργασία (i.e., τοπικό event, αποστολή, λήψη μηνύματος)
  - Κάθε κατάσταση (S0, S1, S2, …) είναι συνεπής

Σειριοποίηση - Linearization

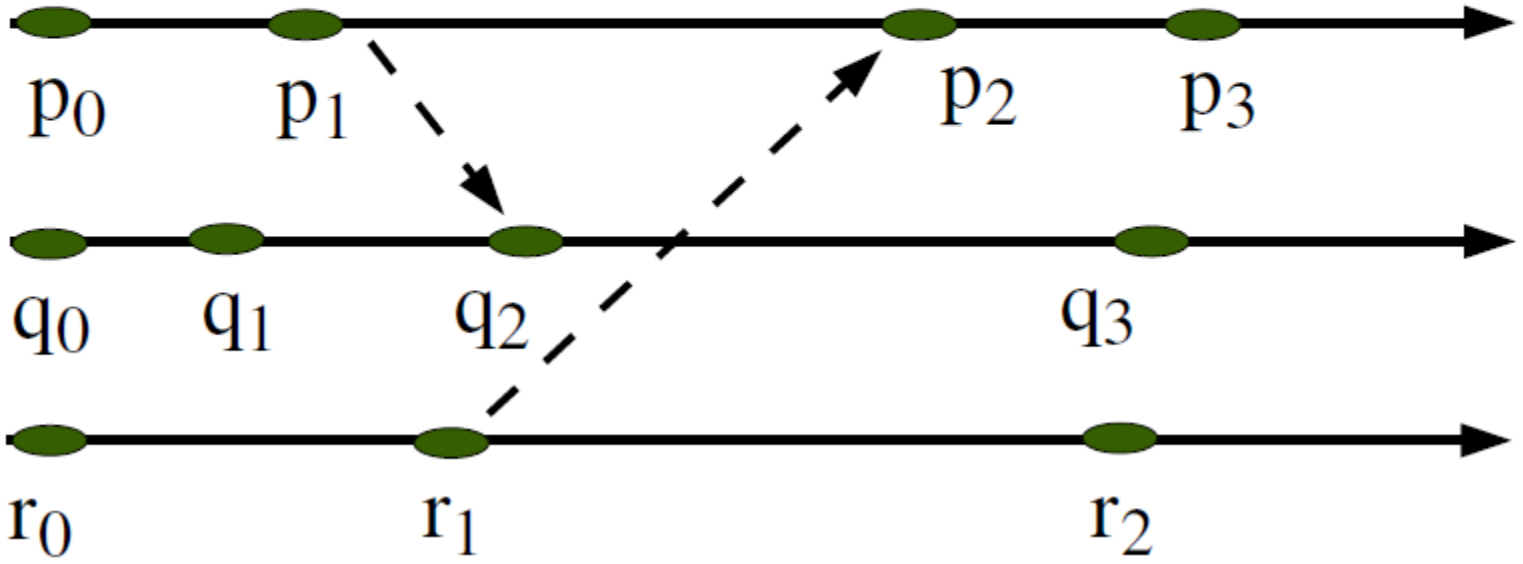
- Μια εκτέλεση (run) είναι μια ολική διάταξη *(total ordering)* όλων των γεγονότων σε ένα global history που είναι συνεπής με κάθε local history.

- Σειριοποίηση (*linearization) ή συνεπής εκτέλεση (consistent run)* είναι μια εκτέλεση που περιγράφει μεταβάσεις ανάμεσα σε consistent global states.

- Ένα state S' είναι προσβάσιμο (reachable) από το state S αν υπάρχει linearization από το S στοS'.

Linearization

$\{p_0, p_1, q_0, r_0, q_1, r_1, p_2, p_3, q_2, r_2, q_3\}$



Γιατί είναι σημαντικό;

- Αν συγκεντρώσουμε όλα τα events και γνωρίζουμε τις σχέσεις *happened before,* τότε μπορούμε να κατασκευάσουμε όλα τα πιθανά linearizations.

- Γνωρίσουμε ότι η πραγματική εκτέλεση πήρε ένα από αυτά τα μονοπάτια

- Μπορούμε να εξάγουμε πληροφορία για την εκτέλεση ακόμα κι αν δεν ξέρουμε ποιο ακριβώς μονοπάτι ακολουθήθηκε

Global state predicates

- Predicate: μια συνάρτηση που παίρνει τις τιμές {true, false} για μια καθολική κατάσταση
  - Σταθερό - stable: άπαξ και γίνει true, παραμένει για το υπόλοιπο της εκτέλεσης, π.χ. deadlock.
  - Ασταθές non-stable: μπορεί να γίνει true και μετά false, π.χ., αν οι τιμές 2 αντιγράφων είναι συνεπείς

Ιδιότητες ενός predicate

- Liveness (of a predicate): η εγγύηση ότι κάτι καλό θα συμβεί τελικά
  - Για κάθε σειριοποίηση ξεκινώντας από την αρχική κατάσταση, υπάρχει μια προσπελάσιμη κατάσταση όπου το predicate γίνεται true.
  - Η εγγύηση τερματισμού προσφέρει liveness
- Safety (of a predicate): η εγγύηση ότι κάτι κακό δε θα συμβεί ποτέ
  - Για κάθε κατάσταση προσπελάσιμη από την αρχική, το predicate είναι false.
  - Οι αλγόριθμοι αποφυγής αδιεξόδου Deadlock προσφέρουν ασφάλεια

Ο Αλγόριθμος Snapshot

- *Υποθέσεις:*
  - Υπάρχει δίαυλος επικοινωνίας ανάμεσα σε κάθε ζεύγος διεργασιών
  - Οι δίαυλοι επικοινωνίας είναι αμφίδρομοι και FIFO
  - Όλα τα μηνύματα φτάνουν ακέραια, ακριβώς μια φορά
  - Οποιαδήποτε διεργασία μπορεί να ξεκινήσει τον αλγόριθμο
  - Ο αλγόριθμος δεν παρεμβαίνει στην κανονική λειτουργία του συστήματος
  - Κάθε διεργασία μπορεί να καταγράφει την κατάστασή της και την κατάσταση των εισερχόμενων διαύλων της

Ο Αλγόριθμος Snapshot

- Σκοπός: Η καταγραφή των διεργασιών και των καταστάσεων των διαύλων επικοινωνίας ώστε ο συνδυασμός αυτός να αποτελεί μια συνεπή ολική κατάσταση
- 2 θέματα:
  - #1: Πότε να καταγράψει κάθε διεργασία τοπικό snapshot ώστε το σύνολό τους να αποτελέσει συνεπή ολική κατάσταση;
  - #2: Πώς καταγράφονται τα μηνύματα που ήταν εν κινήσει πριν κάθε τοπικό snapshot?

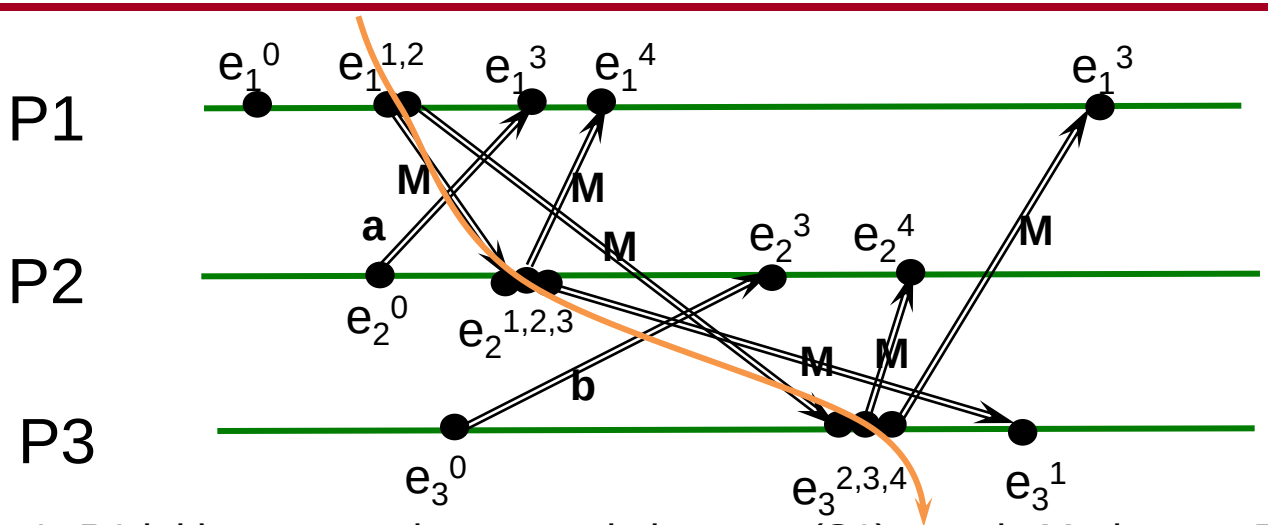
Ο Αλγόριθμος Snapshot

- Βασική ιδέα: broadcast ενός marker και καταγραφή
  - Η διεργασία που ξεκινά τον αλγόριθμο στέλνει με broadcasts ένα μήνυμα “marker” σε όλους («Καταγράψτε ένα τοπικό snapshot»)
  - Αν μια διεργασία λάβει marker για πρώτη φορά, καταγράφει το τοπικό snapshot, αρχίζει να καταγράφει όλα τα εισερχόμενα μηνύματα και στέλνει με broadcast έναν marker πάλι σε όλους (“Έστειλα όλα τα μηνύματα σε εσάς πριν καταγράψω την κατάστασή μου, οπότε σταματήστε να καταγράφετε τα μηνύματά μου”)
  - Μια διεργασία σταματάει να καταγράφει όταν λάβει ένα marker για κάθε δίαυλο επικοινωνίας

Ο αλγόριθμος Chandy και Lamport

- *Marker receiving rule for process p_i*
  - On p_i's receipt of a *marker* message over channel *c*:
  - *if* (p_i has not yet recorded its state) it
    - records its process state now;
    - records the state of *c* as the empty set;
    - turns on recording of messages arriving over other incoming channels;
  - *else*
    - p_i records the state of *c* as the set of messages it has received over *c*
    - since it saved its state.
  - *end if*
- *Marker sending rule for process p_i*
  - After p_i has recorded its state, for each outgoing channel *c*:
  - p_i sends one marker message over *c*
  - (before it sends any other message over *c*).

Άσκηση



1- P1 initiates snapshot: records its state (S1); sends Markers to P2 & P3; turns on recording for channels C21 and C31

2- P2 receives Marker over C12, records its state (S2), sets state(C12) = {} sends Marker to P1 & P3; turns on recording for channel C32

3- P1 receives Marker over C21, sets state(C21) = {a}

4- P3 receives Marker over C13, records its state (S3), sets state(C13) = {} sends Marker to P1 & P2; turns on recording for channel C23

5- P2 receives Marker over C32, sets state(C32) = {b}

6- P3 receives Marker over C23, sets state(C23) = {}

7- P1 receives Marker over C31, sets state(C31) = {}

Ιδιότητα 1

- Ο αλγόριθμος snapshot δίνει consistent cut
- Δηλαδή,
  - Αν e_i είναι ένα event στο P_i, και e_j είναι ένα event στο P_j
  - Αν $e_i \rightarrow e_j$, και e_j είναι στο cut, τότε και το e_i θα είναι στο cut.
- Απόδειξη
  - Ας υποθέσουμε ότι το e_j ανήκει στο cut, αλλά το e_i όχι
  - Επειδή $e_i \rightarrow e_j$, πρέπει να υπάρχει αλληλουχία μηνυμάτων M που οδηγεί στη σχέση αυτή
  - Επειδή το e_i δεν είναι στο cut (από την υπόθεση), θα πρέπει να έχει σταλεί marker πριν το e_i και πριν από όλα τα M.
  - Τότε το P_j θα πρέπει να έχει καταγράψει την κατάστασή του πριν το e_j, δηλ το e_j δεν ανήκει στο cut.

Ιδιότητα 2

- Ένα stable predicate που είναι true σε κάποιο snapshot S-snap πρέπει να είναι true και στο τελικό snapshot S-final (reachability)
  - S-snap: η καταγεγραμμένη καθολική κατάσταση
  - S-final: Η καθολική κατάσταση αμέσως μετά την καταγραφή της τελικής
    - garbage collection
    - dead-lock detection

- Ένα non-stable predicate μπορεί να είναι true σε ένα snapshot αλλά όχι απαραίτητα κατά τη διάρκεια της πραγματικής εκτέλεσης

Possibly and definitely

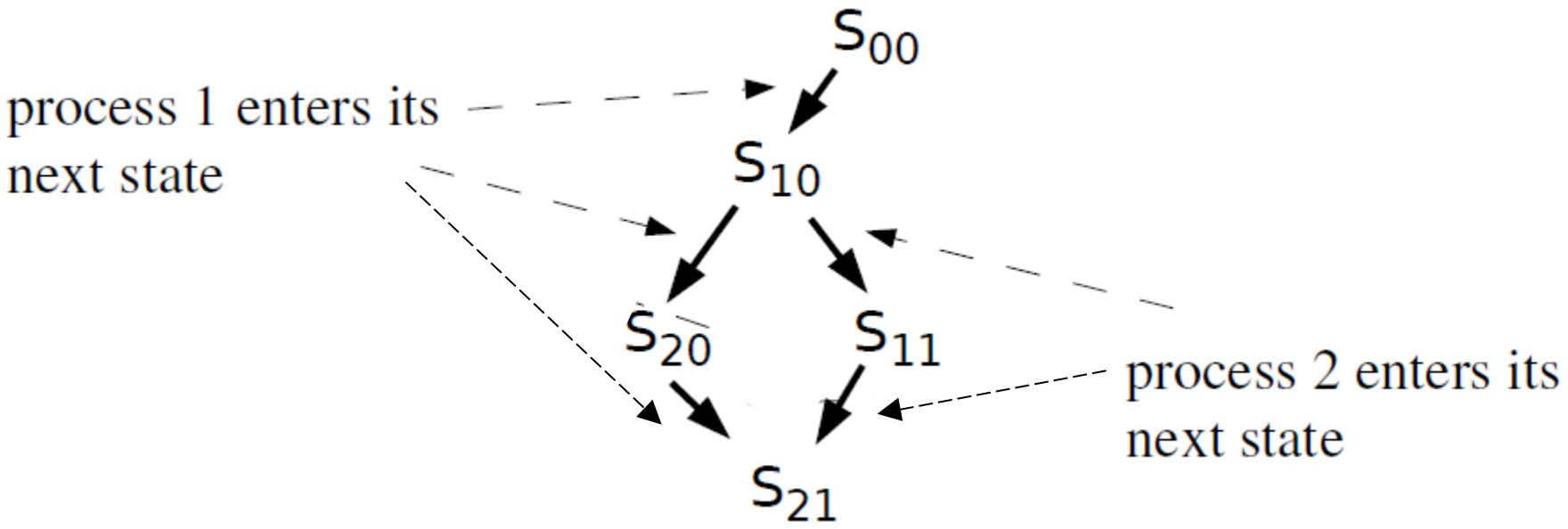
- Θέλουμε να γνωρίζουμε αν ένα non-stable predicate ενδεχομένως συνέβη (possibly) ή σίγουρα συνέβη (definitely)
- Possibly: Υπάρχει consistent global state S από το οποίο περνάει ένα linearization και για το οποίο είναι true ένα predicate ϕ
- Definitely: Για όλα τα linearizations υπάρχει consistent global state S από το οποίο περνάνε και και για το οποίο είναι true ένα predicate ϕ
- Πρέπει να κοιτάξουμε όλες τις πιθανές καταστάσεις
  - Πώς τις κατασκευάζουμε;
  - Είναι αρκετό το απλό snapshot;

Vector timestamped state

- Όταν μια διεργασία i καταγράφει και στέλνει την κατάστασή της, si, στέλνει και το vector timestamp αυτής , V(si)
- Οι καταστάσεις συλλέγονται και ομαδοποιούνται σε global states S = {s0,... sn}.
- Ένα global state είναι consistent iff
  - V(si)[i] ≥ V(sj)[i]
  - Ο αριθμός των events που κατέγραψε η pi για τον εαυτό της είναι πάντα μεγαλύτερος ή ίσος από τον αριθμό των events που κατέγραψαν οι υπόλοιπες διεργασίες για το pi
  - Εξασφαλίζει ότι αν το si εξαρτάται από κάποιο sj τότε το global state θα περιέχει το sj.

Global state lattice

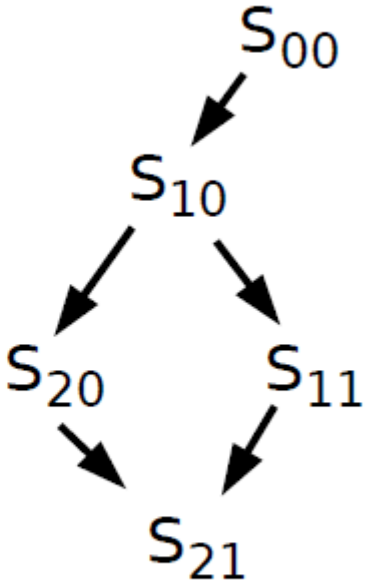
- Το σύνολο των consistent global states φτιάχνουν ένα πλέγμα όπου κάθε ακμή είναι μια πιθανή μετάβαση



Possibly true

$p(S_{20}) = \text{true}$

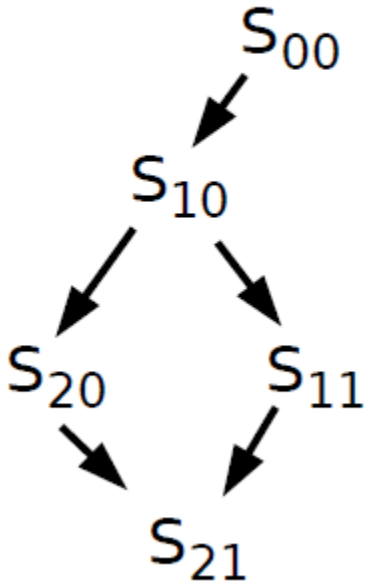
Αν ένα predicate είναι *true σε οποιαδήποτε consistent* global state του πλέγματος τότε είναι *possibly true* κατά την εκτέλεση



Definitively true

Αν δεν υπάρχει μονοπάτι από την αρχική στην τελική κατάσταση χωρίς να περνάει από μια κατάσταση που να είναι true για ένα predicate τότε το predicate είναι *definitely true* κατά την εκτέλεση

$p(S_{20}) = \text{true}$
$p(S_{11}) = \text{true}$



Implementation issues

- What is the state of a process?
  - depends on the predicates we want to evaluate
- How often do we have to send a state report?
  - only when the state changes
- How do we best construct the lattice and can we monitor it during execution?
  - yes we can incrementally construct global states and build the lattice

Τι είδαμε σήμερα

- Ο συγχρονισμός απαραίτητος για τα κατανεμημένα συστήματα
  - Αλγόριθμος Cristian
  - Αλγόριθμος Berkeley
  - NTP
- Σχετική διάταξη των γεγονότων αρκετή πρακτικά
  - Lamport's logical clocks
  - Vector clocks
- Καθολικές καταστάσεις
  - Η ένωση των καταστάσεων όλων των διεργασιών
  - Συνεπής ολική κατάσταση vs. Ασυνεπής ολική κατάσταση
- Ο αλγόριθμος "snapshot"
  - Πάρε ένα snapshot της τοπικής κατάστασης
  - Broadcast ενός "marker" μηνύματος για να ξεκινήσουν και οι υπόλοιπες διεργασίες την καταγραφή
  - Αρχίζει καταγραφή όλων των εισερχόμενων μηνυμάτων από όλα τα κανάλια μιας διεργασίας μέχρι τη λήψη του επόμενου "marker"
  - Το αποτέλεσμα: συνεπής καθολική κατάσταση