Systèmes Distribués

Licence Informatique 3ème année

Gestion du temps & état global dans un système distribué

Eric Cariou

Université de Pau et des Pays de l'Adour
UFR Sciences Pau – Département Informatique

Eric.Cariou@univ-pau.fr

Introduction

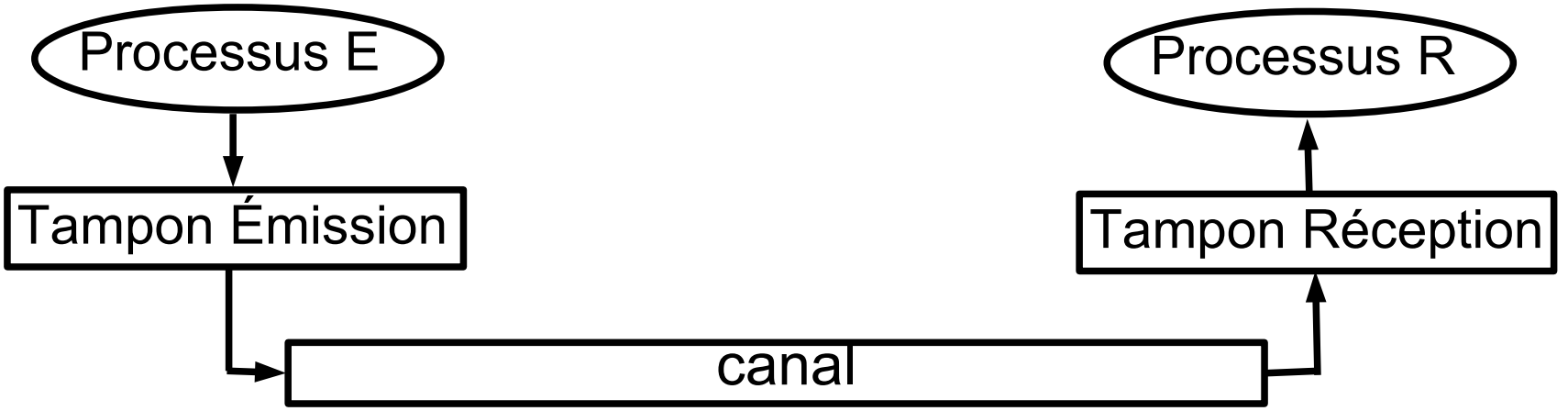
- Algorithmique distribuée
  - Développement d'algorithmes dédiés aux systèmes distribués et prenant en compte les spécificités de ces systèmes
  - On y retrouve notamment des adaptations de problèmes classiques en parallélisme
    - Exclusion mutuelle, élection d'un maître ...
  - Mais aussi des problèmes typiques des systèmes distribués
    - Horloge globale, état global, diffusion causale, consensus ...
  - S'intéresse principalement à deux grandes familles de problèmes
    - Synchronisation et coordination entre processus distants
    - Entente sur valeurs communes et cohérence globale dans un contexte non fiable (crash de processus, perte de messages ...)
- Dans ce cours
  - Introduction à l'algorithmique distribuée avec les problèmes d'horloge et d'état globaux

Introduction

- Contraintes particulières des systèmes distribués à prendre en compte
  - Concurrence
    - Les éléments formant le système s'exécutent en parallèle et de manière autonome
    - Pas d'état ou d'horloge globale commune
  - Points de problèmes de fiabilité en nombre accru
    - Problème matériel d'une machine
    - Problème de communication via le réseau
    - Problème logiciel sur un des éléments du système
  - Communication est un point crucial
    - Potentiellement non fiable
    - Temps de communication non négligeables

Système distribué

- Système distribué
  - Ensemble d'éléments logiciels s'exécutant en parallèle
    - On parlera de processus pour ces éléments logiciels
  - Ces éléments communiquent via des canaux de communication
    - Liaison entre un processus et un canal (canal unidirectionnel ici)



Processus

- Processus
  - Élément logiciel effectuant une tâche, un calcul
    - Exécution d'un ensemble d'instructions
    - Une instruction correspond à un événement local au processus
      - Dont les événements d'émission et de réception de messages
    - Les instructions sont généralement considérées comme atomiques
  - Il possède une mémoire locale
  - Il possède un état local
    - Ensemble de ses données et des valeurs de ses variables locales
  - Il possède un identifiant qu'il connaît
    - Et connaît en général les identifiants des/d'autres processus
  - Pas de connaissance sur l'état des autres processus
  - Les processus d'un système s'exécutent en parallèle

Canaux

- Canal de communication
  - Canal logique de communication point à point
    - Pour communication entre 2 processus
    - Transit de messages sur un canal
  - Caractéristiques d'un canal
    - Uni ou bi-directionnel
    - Fiable ou non : perd/modifie ou pas des messages
    - Ordre de réception par rapport à l'émission
      - Exemple : FIFO = les messages sont reçus dans l'ordre où ils sont émis
    - Synchrone ou asynchrone
      - Synchrone : l'émetteur et le récepteur se synchronisent pour réaliser l'émission et/ou la réception
      - Asynchrone : pas de synchronisation entre émetteur et récepteur
    - Taille des tampons de message cotés émetteur et récepteur
      - Limitée ou illimitée

Canaux

- Caractéristiques d'un canal
  - Modèle généralement utilisé
    - Fiable, FIFO, tampon de taille illimitée, asynchrone (en émission et réception) et bidirectionnel
    - Variante courante avec réception synchrone
  - Exemple : modèle des sockets TCP
    - Fiable
    - FIFO
    - Bidirectionnel
    - Synchrone pour la réception
      - On reçoit quand l'émetteur émet
      - Sauf si données non lues dans le tampon coté récepteur
    - Asynchrone en émission
      - Émetteur n'est pas bloqué quand il émet quoique fasse le récepteur

Système synchrone / asynchrone

- Un modèle synchrone est un modèle où les contraintes temporelles sont bornées
  - On sait qu'un processus évoluera dans un temps borné
  - On sait qu'un message arrivera en un certain délai
  - On connaît la limite de dérive des horloges locales
- Un modèle asynchrone n'offre aucune borne temporelle
  - Modèle bien plus contraignant et rendant impossible ou difficile la réalisation de certains algorithmes distribués
    - Exemple : ne sait pas différencier en asynchrone
      - Le fait qu'un processus est lent ou est planté
      - Du fait qu'un message est long à transiter ou est perdu

État et horloge globales

- Pour chaque processus du système
  - État local : valeur des variables du processus à un instant *t*
- État global du système
  - Valeur de toutes les variables de tous les processus du système à un instant *t*
- Problème
  - Un état est lié à un instant *t*
  - Mais
    - Chaque processus à une horloge physique locale
    - Pas d'horloge globale dans un système distribué
- La définition d'un état global est possible seulement si on est capable de définir un temps global

Temps

- Définir un temps global cohérent et « identique » (ou presque) pour tous les processus
  - Soit synchroniser au mieux les horloges physiques locales avec une horloge de référence ou entre elles
  - Soit créer un temps logique
- Synchronisation des horloges physiques locales
  - But est d'éviter qu'une horloge locale dérive trop par rapport à un référentiel de temps
    - La dérive est bornée en augmentation et en diminution
  - Deux modes
    - Synchronisation interne
    - Synchronisation externe

Synchronisation horloges physiques

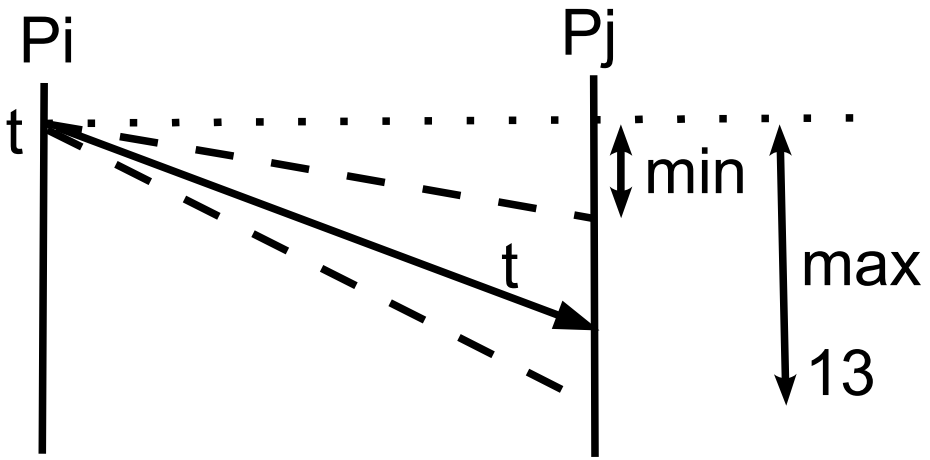
- Synchronisation interne
  - Un groupe d'horloges ont entre elles une dérive bornée
  - Si *D* est la borne max de la dérive, C_i l'horloge physique du processus *i*, *t* un instant réel, et $C_i(t)$ la valeur de l'horloge C_i pour cet instant *t,* alors
    - $\forall i,j,t : | C_i(t) - C_j(t) | < D$
    - A tout instant, la dérive entre 2 horloges quelconques du groupe ne dépasse jamais la borne D
      - Une mesure locale du temps sur un site donnera un temps identique à une mesure d'un autre site avec une différence d'au plus D
  - Les dérives sont bornées entre les horloges mais pas nécessairement avec le temps réel
    - Sauf si synchronisation externe en plus sur un temps de référence

Synchronisation horloges physiques

- Synchronisation externe
  - Une source externe fournit un temps de référence
  - Les horloges locales se re-synchronisent régulièrement à partir de cette source, en prenant en compte
    - Les temps de propagation des messages entre le processus courant et celui gérant le temps de référence
    - Le temps de traitement de la requête (récupérer la date de référence) du processus gérant le temps de référence, s'il est connu
  - Même contrainte que pour synchronisation interne : dérive bornée
    - Mais par rapport à une source de référence : S
      - A l'instant *t*, *S(t)* est le temps de référence donné par la source externe
      - $\forall i,t : | S(t) - C_i(t) | < D$

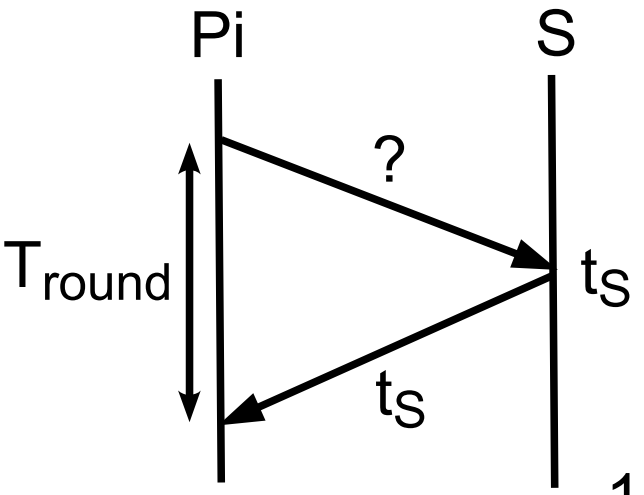
Méthodes de synchronisation

- Synchronisation interne pour un système distribué synchrone
  - Un processus envoie régulièrement la valeur *t* de son horloge aux autres processus pour que chacun se recale sur l'horloge des autres
  - En théorie : processus met son horloge à la valeur t + T_{trans}
    - T_{trans} = temps de transmission du message mais est non fixe
    - T_{trans} est borné : borne max mais aussi borne min : min < T_{trans} < max
  - En pratique, processus met son horloge à la valeur de t + x
    - Où *x* est une estimation du temps de propagation
      - Exemple : on prend la moyenne des bornes, x = (min + max) / 2
      - Erreur de recalage est alors au plus de (min – max) / 2
- Plus difficile à mettre en œuvre dans un système asynchrone à cause des temps de transmission non bornés



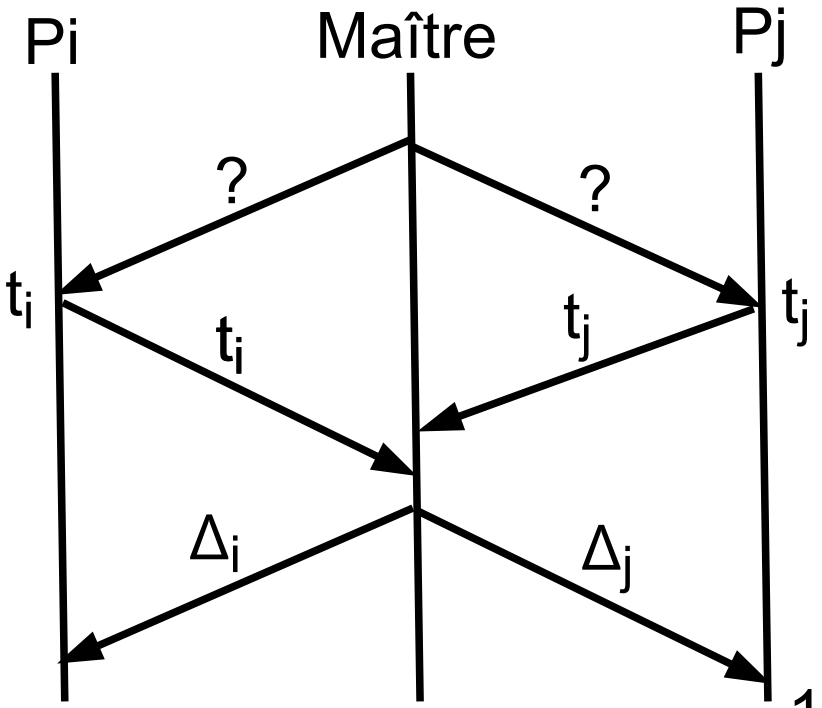
Méthodes de synchronisation

- Synchronisation externe pour un système asynchrone
  - Temps de propagation des messages est non borné
  - Mais temps d'aller-retour entre 2 processus est mesurable
    - Un processus envoie une requête au processus gérant le temps de référence qui lui répond en lui envoyant sa valeur d'horloge t_s
      - Mesure du temps d'aller-retour : T_{round}
    - A partir de là, on peut mettre à jour son horloge
      - En prenant la moitié de T_{round} par exemple
        - $t = t_s + T_{round}/2$
    - Certains algorithmes se basent également sur la durée du temps d'aller-retour par rapport à la précision requise pour savoir si la mise-à-jour sera assez précise [Cristian, 89]



Méthodes de synchronisation

- Synchronisation interne pour un système asynchrone [Gusella et Zatti, 89]
  - Un des processus est élu maître et c'est lui qui gère le temps
  - Les autres processus lui envoient leur valeur d'horloge
    - En utilisant également les temps d'aller-retour
  - A partir de toutes les horloges, le maître détermine le temps moyen global
    - Il envoie à chaque processus *la variation* de leur temps local par rapport au temps global
      - Pas d'influence du temps de propagation du message
    - Permet d'éliminer les horloges défaillantes car le maître a une vision globale
  - Si le maître se plante, un autre est élu



Méthodes de synchronisation

- Méthodes/algorithmes de Guzella et Christian
  - Plutôt adaptés aux réseaux locaux
- NTP : Network Time Protocol
  - Reprend les idées de ces 2 méthodes mais pour fournir un temps global via Internet
  - Adapté pour résister aux problèmes des larges réseaux
    - Redondance des serveurs de temps pour assurer fiabilité
      - Se base sur un réseau de serveurs
    - Permet à n'importe quel client, quelque soit la qualité de la communication de se synchroniser
    - Authentification pour se protéger des attaques

Temps logique

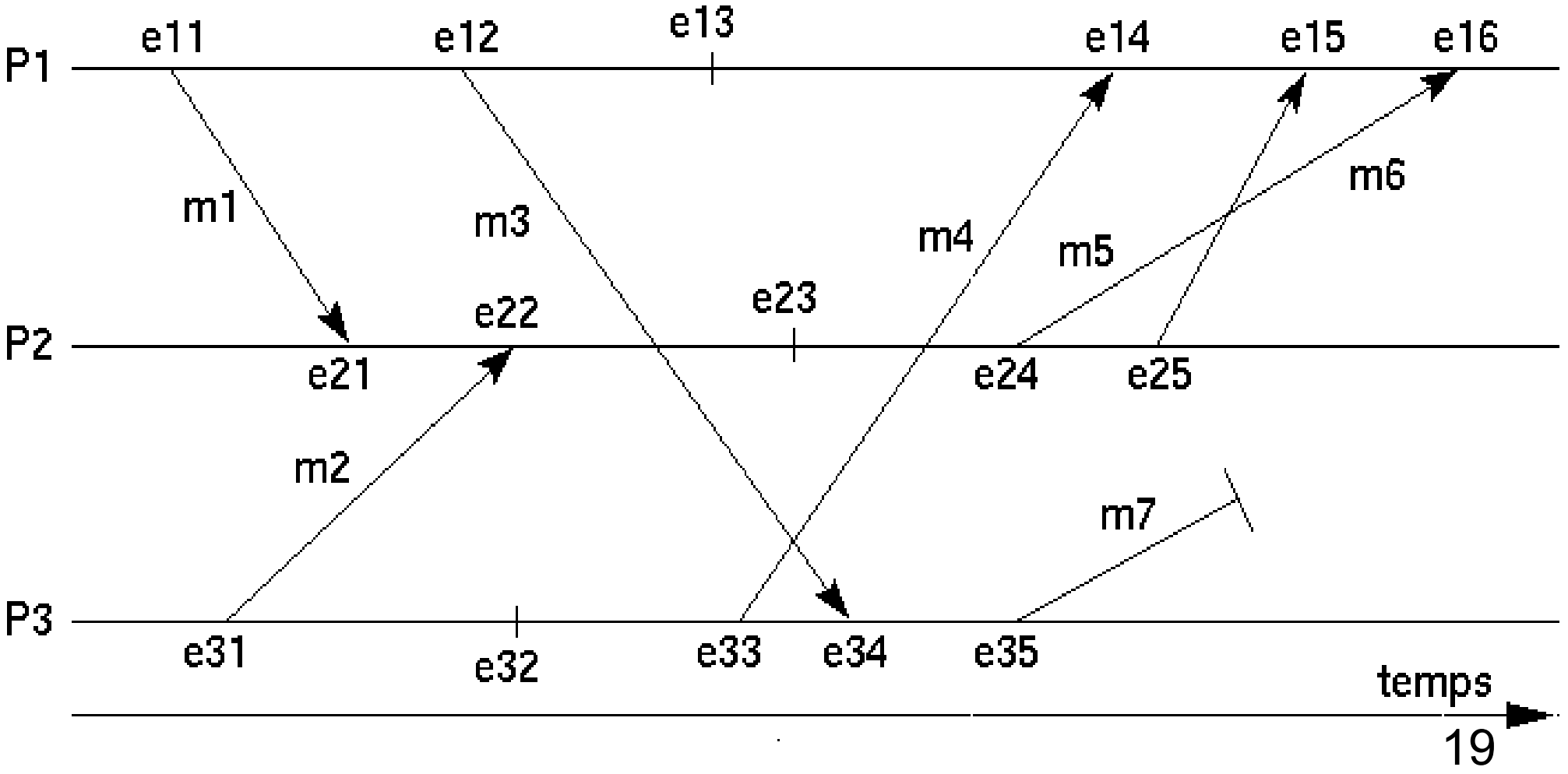
- Temps logique
  - Temps qui n'est pas lié à un temps physique
  - But est de pouvoir préciser l'ordonnancement de l'exécution des processus et de leur communication
  - En fonction des événements locaux des processus, des messages envoyés et reçus, on créé un ordonnancement logique
    - Création horloge logique
- Deux approches principales
  - Horloge de Lamport : méthode par estampille
  - Horloge de Mattern : horloge vectorielle

Temps logique : chronogramme

- Chronogramme
  - Décrit l'ordonnancement temporel des événements des processus et des échanges de messages
  - Chaque processus est représenté par une ligne
  - Trois types d'événements signalés sur une ligne
    - Émission d'un message à destination d'un autre processus
    - Réception d'un message venant d'un autre processus
    - Événement interne dans l'évolution du processus
  - Les messages échangés doivent respecter la topologie de liaison des processus via les canaux

Temps logique : chronogramme

- Trois processus tous reliés entre-eux par des canaux
- Temps de propagation des messages quelconques et possibilité de perte de message



Temps logique : chronogramme

- Exemples d'événements
  - Processus P1
    - e11 : événement d'émission du message m1 à destination du processus P2
    - e13 : événement interne au processus
    - e14 : réception du message m4 venant du processus P3
  - Processus P2 : message m5 envoyé avant m6 mais m6 reçu avant m5
  - Processus P3 : le message m7 est perdu par le canal de communication
- Règle de numérotation d'un événement
  - e_{xy} avec x le numéro du processus et y le numéro de l'événement pour le processus, dans l'ordre croissant

Temps logique : dépendance causale

- Relation de dépendance causale
  - Il y a une dépendance causale entre 2 événements si un événement doit avoir lieu avant l'autre
  - Notation : e $\rightarrow$ e'
    - *e* doit se dérouler avant *e′*
  - Si e $\rightarrow$ e', alors une des trois conditions suivantes doit être vérifiée pour *e* et *e′*
    - Si *e* et *e′* sont des événements d'un même processus, *e* précède localement *e′*
    - Si *e* est l'émission d'un message, *e′* est la réception de ce message
    - Il existe un événement *f* tel que e $\rightarrow$ f et f $\rightarrow$ e'

Temps logique : dépendance causale

- Sur exemple précédent
  - Quelques dépendances causales autour de e12
    - Localement : e11 $\rightarrow$ e12, e12 $\rightarrow$ e13
    - Sur message : e12 $\rightarrow$ e34
    - Par transitivité : e12 $\rightarrow$ e35 (car e34 $\rightarrow$ e35) et e11 $\rightarrow$ e13
  - Dépendance causale entre e12 et e32 ?
    - A priori non : absence de dépendance causale
    - Des événements non liés causalement se déroulent en parallèle
- Relation de parallélisme : ||
  - e || e' $\Leftrightarrow$ ¬((e $\rightarrow$ e') v (e' $\rightarrow$ e))
  - Parallélisme logique : ne signifie pas que les 2 événements se déroulent simultanément mais qu'il peuvent se dérouler dans n'importe quel ordre

Temps logique : dépendance causale

- Ordonnancement des événements
  - Les dépendances causales définissent des ordres partiels pour des ensembles d'événements
- Buts d'une horloge logique (selon le type d'horloge)
  - Créer un ordre total global sur tous les événements de tous les processus
  - Déterminer si un événement a eu lieu avant un autre ou s'il n'y a pas de dépendances causales entre eux
  - Vérifier que des propriétés d'ordre sur l'arrivée de messages sont respectées
- Horloge logique
  - Fonction H(e) : associe une date à chaque événement
  - Respect des dépendances causales

Horloge de Lamport

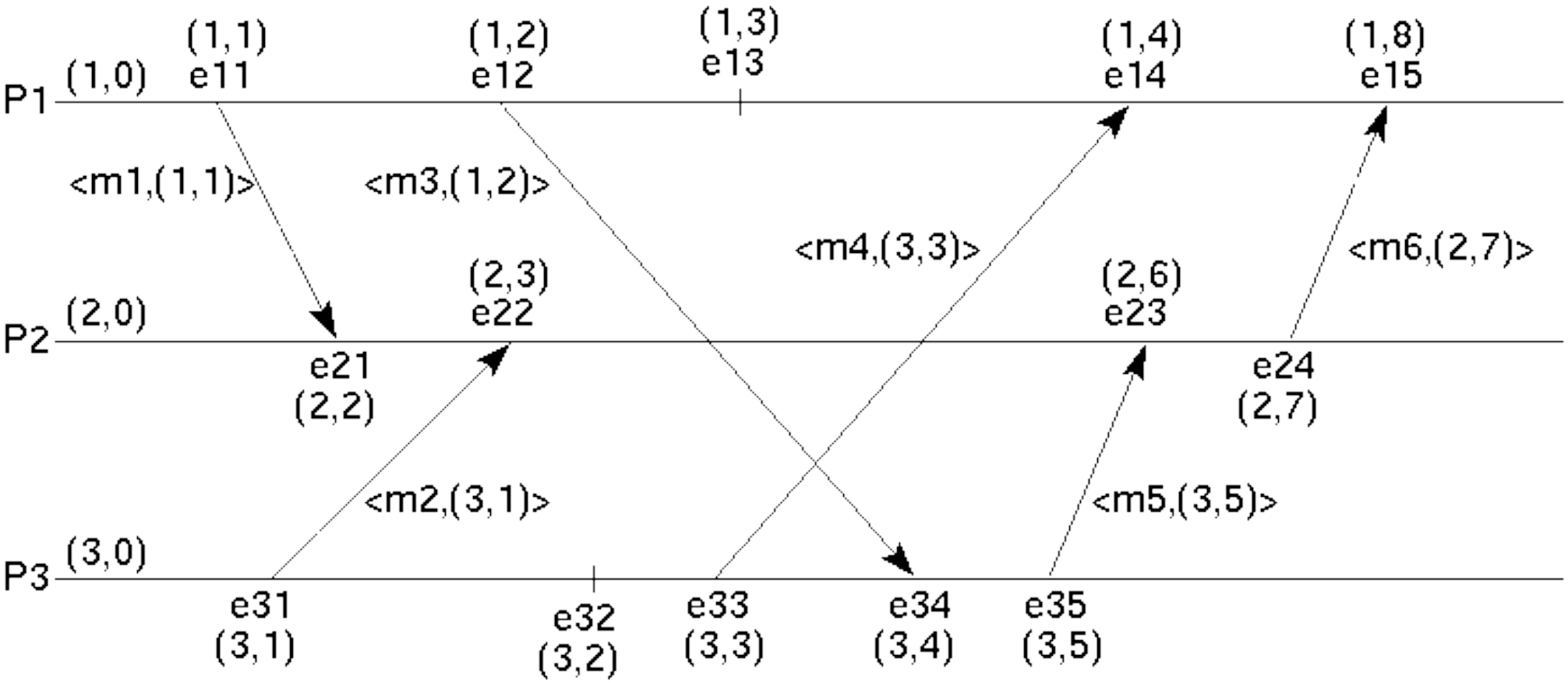
- Horloge de Lamport, 1978
  - A chaque événement *e*, une date H(e) renvoie un couple (s, nb)
    - s : numéro du processus
    - nb : numéro d'événement (estampille)
  - Unicité des dates : pas le même couple (s,nb) pour deux événements différents
    - Implique que sur un même processus, deux événements différents n'ont pas la même estampille
    - $\forall d,d' : d.s = d'.s \Rightarrow d.nb \neq d'.nb$
  - Deux dates sont toujours ordonnées
    - $H(e) < H(e') \Rightarrow (e.nb < e'.nb)$ ou ($(e.nb=e'.nb)$ et $(e.s < e'.s)$)
  - Horloges de Lamport respectent les dépendances causales
    - $e \rightarrow e' \Rightarrow H(e) < H(e')$
    - La réciproque n'est pas vraie
      - $H(e) < H(e') \Rightarrow \neg (e' \rightarrow e)$
      - C'est-à-dire : soit $e \rightarrow e'$, soit $e \parallel e'$

Horloge de Lamport

- Création du temps logique
  - Localement, chaque processus Pi possède une horloge locale logique (estampille) Hi, initialisée à 0
    - Sert à dater les événements
  - Pour chaque événement local de Pi
    - Hi = Hi + 1 : on incrémente l'horloge locale
    - L'événement a pour date (i,Hi)
  - Émission d'un message par Pi
    - On incrémente Hi de 1 puis on envoie le message en envoyant la date (i, Hi) de l'événement d'émission avec le message
  - Réception d'un message *m* avec la date (s, nb)
    - Hi = max(Hi, nb) +1 et marque l'événement de réception avec (i,Hi)
      - Hi est recalée sur l'horloge de l'autre processus s'il est « en avance » avant d'être incrémentée

Horloge de Lamport

- Exemple : chronogramme avec ajouts des estampilles



- Date de e23 : 6 car le message m5 reçu avait une valeur de 5 et l'horloge locale est seulement à 3
- Date de e34 : 4 car on incrémente l'horloge locale vu que sa valeur est supérieure à celle du message m3
- Pour e11, e12, e13 ... : incrémentation de +1 de l'horloge locale

Horloge de Lamport

- Ordonnancement global
  - L'intérêt des horloges de Lamport est de créer un ordre global total entre tous les événements du système
    - Ordre respecte les dépendances causales entre événements
    - Pour deux événements indépendants causalement, un choix arbitraire est fait entre les deux
      - Ne pose pas de problème car pas de contraintes de dépendances à respecter
  - Ordre total, noté e << e' : *e* s'est déroulé avant *e'*
    - Soit *e* événement de *Pi* et *e'* événement de *Pj* :
      e << e' $\Leftrightarrow$ Hi(e) < Hj(e')
      - Localement (si i = j), Hi donne l'ordre des événements du processus
      - Les 2 horloges de 2 processus différents permettent de déterminer l'ordonnancement des événements des 2 processus
        - Si égalité de la valeur de l'estampille, le numéro du processus est utilisé pour les ordonner

Horloge de Lamport

- Ordre total global obtenu pour l'exemple
  - e11 << e31 << e12<< e21 << e32 << e13 << e22 << e33 << e14 << e34 << e35 << e23 << e24 << e15
  - D'autres seraient valides mais l'algorithme n'en donne qu'un
- Limites de l'horloge de Lamport
  - Elle respecte les dépendances causales mais avec *e* et *e′* tel que H(e) < H(e') on ne peut rien dire sur la dépendance entre *e* et *e′*
    - Dépendance causale directe ou transitive entre *e* et *e′* ? Ou aucune dépendance causale ?
    - Exemple : H(e32) = 2 et H(e13) = 3 mais on a pas e32$\rightarrow$ e13

Horloge de Mattern

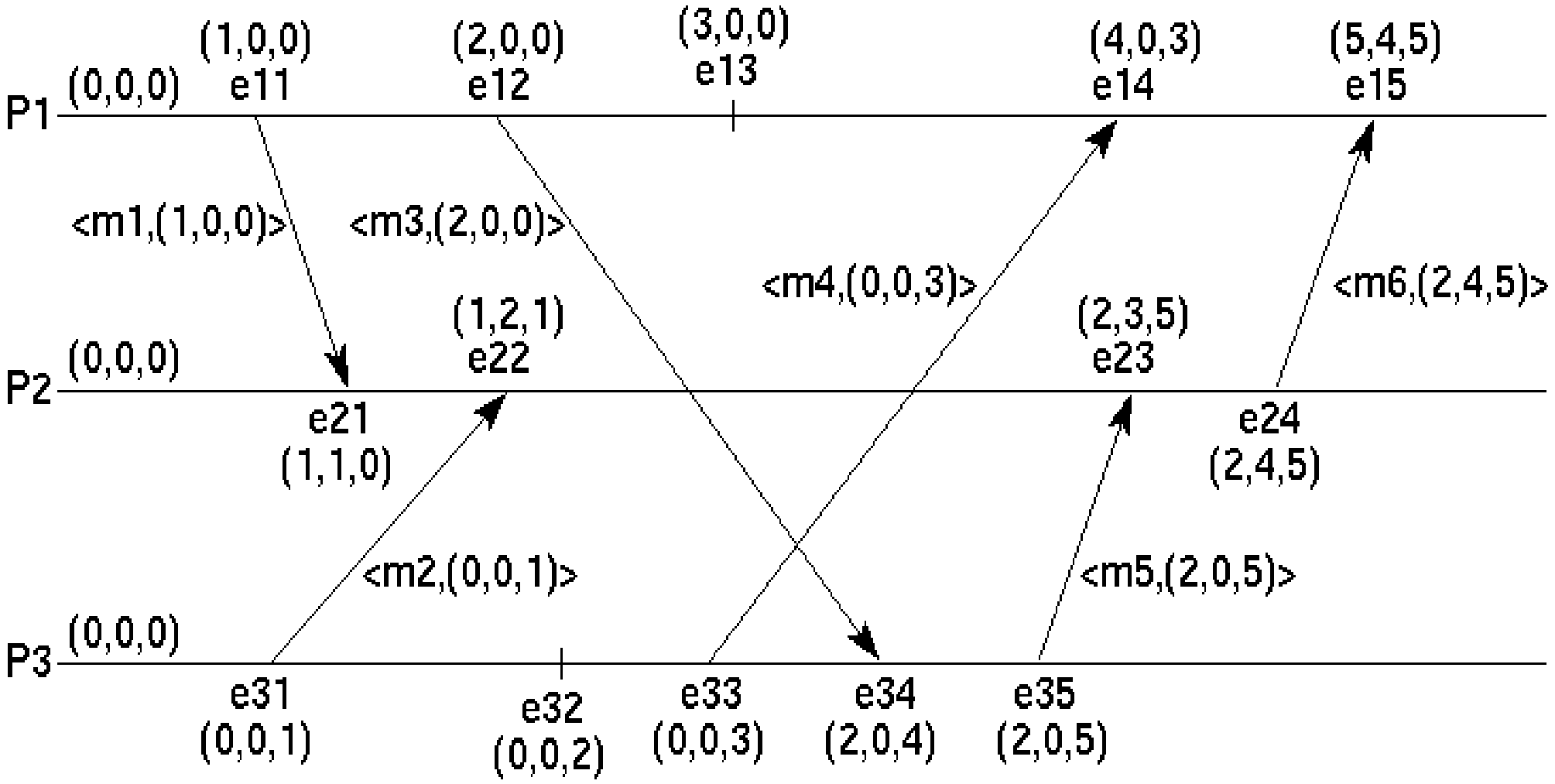
- Horloge de Mattern & Fidge, 1989-91
  - Horloge qui assure la réciproque de la dépendance causale
    - $H(e) < H(e') \Rightarrow e \rightarrow e'$
  - Permet également de savoir si 2 événements sont parallèles (non dépendants causalement)
  - Ne définit par contre pas un ordre total global
- Principe
  - Chaque événement est là aussi daté par une horloge
    - Date = vecteur V de taille égale au nombre de processus
  - Localement, chaque processus *Pi* a un vecteur *Vi*
  - Pour chaque processus *Pi*, chaque case *Vi[j]* du vecteur contiendra des valeurs de l'horloge du processus *Pj*

Horloge de Mattern

- Fonctionnement de l'horloge
  - Initialisation : pour chaque processus *Pi*, Vi =(0, ... , 0)
  - Pour un processus *Pi*, à chacun de ses événements (local, émission, réception) :
    - Vi [i] = Vi [i] + 1
    - Incrémentation du compteur local d'événement
    - Si émission d'un message, alors *Vi* est envoyé avec le message
  - Pour un processus *Pi*, à la réception d'un message *m* contenant un vecteur *Vm*, on met à jour les cases $j \neq i$ de son vecteur local *Vi*
    - $\forall j$: Vi [j] = max (Vm [j], Vi [j])
      - Mémorise le nombre d'événements sur Pj qui sont sur Pj dépendants causalement par rapport à l'émission du message
      - La réception du message est donc aussi dépendante causalement de ces événements sur Pj

Horloge de Mattern

- Exemple : chronogramme d'application horloge de Mattern
  - Même exemple que pour horloge de Lamport



Horloge de Mattern

- Relation d'ordre partiel sur les dates
  - $V \leq V'$ défini par $\forall i : V[i] \leq V'[i]$
  - $V < V'$ défini par $V \leq V'$ et $\exists j$ tel que $V[j] < V'[j]$
  - $V \,||\, V'$ défini par $\neg(V < V') \wedge \neg(V' < V)$
- Dépendance et indépendance causales
  - Horloge de Mattern assure les propriétés suivantes, avec *e* et *e'* deux événements et *V(e)* et *V(e')* leurs datations
    - $V(e) < V(e') \Rightarrow e \rightarrow e'$
      - Si deux dates sont ordonnées, on a forcément une dépendance causale entre les événements datés
    - $V(e) \,||\, V(e') \Rightarrow e \,||\, e'$
      - Si il n'y a aucun ordre entre les 2 dates, les 2 événements sont indépendants causalement

Horloge de Mattern

- Retour sur l'exemple
  - V(e13) = (3,0,0), V(e14) = (4,0,3), V(e15) = (5,4,5)
    - V(e13) < V(e14) donc e13 $\rightarrow$ e14
    - V(e14) < V(e15) donc e13 $\rightarrow$ e15
  - V(e35) = (2,0,5) et V(e23) = (2,3,5)
    - V(e35) < v(e23) donc e35 $\rightarrow$ e23
  - L'horloge de Mattern respecte les dépendances causales des événements
    - Horloge de Lamport respecte cela également
  - V(e32) = (0,0,2) et V(e13) = (3, 0, 0)
    - On a ni V(e32) < V(e13) ni V(e13) < V(e32) donc e32 || e13
    - L'horloge de Mattern détecte les indépendances causales
      - L'horloge de Lamport impose un ordre arbitraire fictif entre les événements indépendants causalement

Horloge de Mattern

- Limite de l'horloge de Mattern
  - Ne permet pas de définir un ordre global total
  - En cas de nombreux processus, la taille du vecteur peut-être importante et donc des données à transmettre relativement importante

État Global

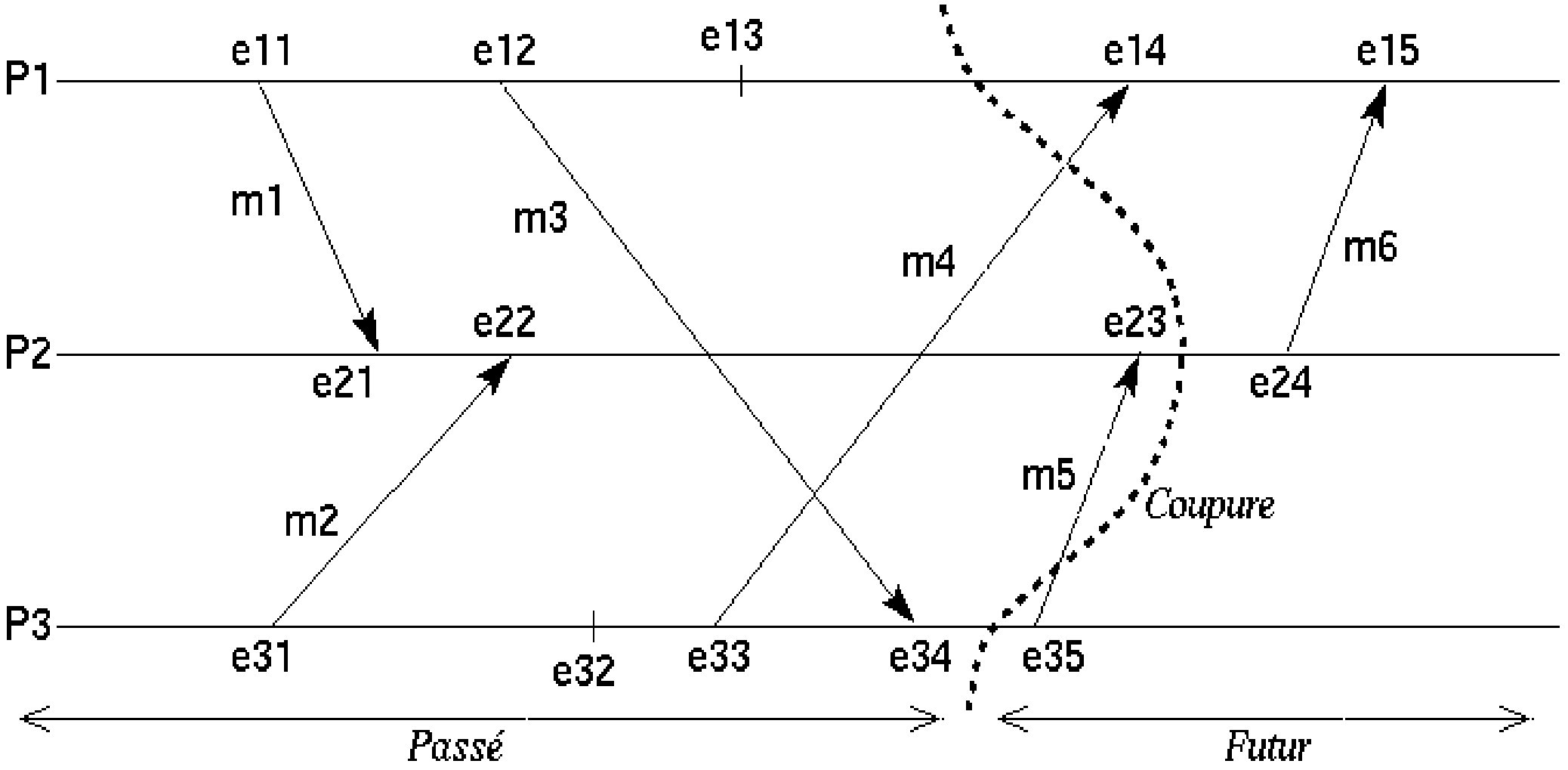
- État global
  - État du système à un instant donné
  - Buts de la recherche d'états globaux
    - Trouver des états cohérents à partir desquels on peut reprendre un calcul distribué en cas de plantage du système
    - Détection de propriétés stables, du respect d'invariants
    - Faciliter le debugging et la mise au point d'applications distribuées
  - Défini à partir de coupures
- Coupure
  - Photographie à un instant donné de l'état du système
  - Définit les événements appartenant au passé et au futur par rapport à l'instant présent de la coupure

Coupure

- Définition coupure
  - Calcul distribué = ensemble *E* d'événements
  - Coupure *C* est un sous-ensemble fini de *E* tel que
    - Soit *a* et *b* deux événements du même processus :
      $a \in C$ et $b \rightarrow a \Rightarrow b \in C$
    - Si un événement d'un processus appartient au passé, alors tous les événements locaux le précédant y appartiennent également
- État associé à une coupure
  - Si le système est composé de N processus, l'état associé à une coupure est défini au niveau d'un ensemble de N événements $(e_1, e_2, \ldots e_i, \ldots e_N)$, avec e_i événement du processus Pi tel que
    - $\forall i : \forall e \in C$ et *e* événement du processus Pi $\Rightarrow e \rightarrow e_i$
    - L'état est défini à la frontière de la coupure : l'événement le plus récent pour chaque processus

Coupure

- Exemple de coupure
  (même chronogramme que pour exemples horloges Lamport et Mattern)



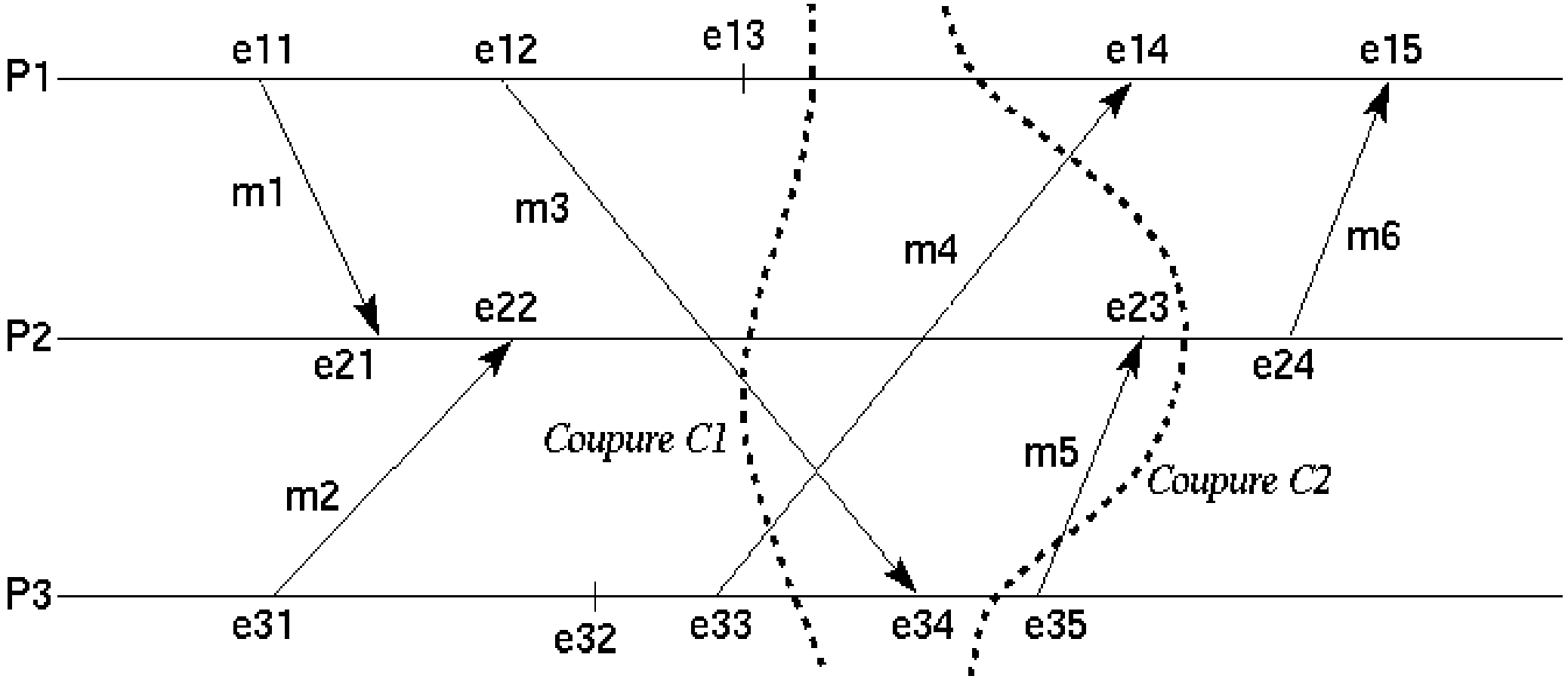
- Coupure = ensemble { e11, e12, e13, e21, e22, e23, e31, e32, e33, e34 }
- État défini par la coupure = (e13, e23, e34)

Coupure/état cohérent

- Coupure cohérente
  - Coupure qui respecte les dépendances causales des événements du système
    - Et pas seulement les dépendances causales locales à chaque processus
    - Soit *a* et *b* deux événements du système :
      $a \in C$ et $b \rightarrow a \Rightarrow b \in C$
    - Coupure cohérente : aucun message ne vient du futur
- État cohérent
  - État associé à une coupure cohérente
  - Permet par exemple une reprise sur faute

Coupure cohérente

- Exemple (même chronogramme que précédent)



- Coupure C1 : cohérente
- Coupure C2 : non cohérente car e35 $\rightarrow$ e23 mais e35 $\notin$ C2
  - La réception de *m5* est dans la coupure mais pas son émission
  - *m5* vient du futur par rapport à la coupure

Datation Coupure

- Horloge de Mattern permet de dater la coupure
  - Soit N processus, C la coupure, e_i l'événement le plus récent pour le processus Pi, $V(e_i)$ la datation de e_i et V(C) la datation de la coupure
    - $V(C) = \max (V(e_1), \dots , V(e_N)) :$
      $\forall i : V(C)[i] = \max (V(e_1)[i] , \dots , V(e_N)[i])$
    - Pour chaque valeur du vecteur, on prend le maximum des valeurs de tous les vecteurs des N événements pour le même indice
- Permet également de déterminer si la coupure est cohérente
  - Cohérent si $V(C) = (V(e_1)[1], \dots , V(e_i)[i], \dots , V(e_N) [N])$
  - Pour un processus Pi, si l'événement e_i est le plus récent c'est lui qui a la date la plus récente pour C : sinon un événement e_j d'un processus Pj ($i \neq j$) s'est déroulé après un événement e_i' de Pi avec e_i' plus récent que e_i
    - $e_i \rightarrow e_i'$ et $e_i' \rightarrow e_j$ avec $e_i \in C$, $e_j \in C$ et $e_i' \notin C$

Datation Coupure

- Datation des coupures de l'exemple
  - Coupure C1 : état = (e13, e22, e33)
    - V(e13) = (3,0,0), V(e22) = (1,2,1), V(e33) = (0,0,3)
    - V(C) = (max(**3**,1,0), max(0,**2**,0), max(0,1,**3**)) = (3,2,3)
    - Coupure cohérente car V(C)[1] = V(e13)[1], V(C)[2] = V(e22)[2], V(C)[3] = V(e33)[3]
  - Coupure C2 : état = (e13, e23, e34)
    - V(e13) = (3,0,0), V(e23) = (2,3,5), V(e34) = (2,0,4)
    - V(C) = (max(**3**,2,2), max(0,**3**,0), max (0,**5**,4))
    - Non cohérent car V(C)[3] ≠ V(e34)[3]
      - D'après la date de e23, e23 doit se dérouler après 5 événements de P3 or e34 n'est que le quatrième événement de P3
      - Un événement de P3 dont e23 dépend causalement n'est donc pas dans la coupure (il s'agit de e35 se déroulant dans le futur)

Détermination état cohérent

- Algorithme de Chandy & Lamport, 1985
  - Algorithme permettant aux processus distribués d'enregistrer un état global cohérent
- Principe général
  - Un processus diffuse un événement marqueur et les processus enregistrent leur état
    - Fonctionnement asynchrone
- Contraintes sur le système
  - Canaux de communication uni-directionnels et FIFO
  - Fiable : pas de plantage ou de perte de message
  - Topologie de connexion fortement (voire totalement) connexe
    - Pour faciliter la diffusion

Détermination état cohérent

- État d'un canal
  - Pour un canal FIFO fiable unidirectionnel
  - Canal pour envoi de message du processus Pi vers Pj
  - A un instant *t*, l'état du canal < i, j > est l'ensemble des messages émis par Pi et non encore reçus par Pj
  - Pour le chronogramme de l'exemple, états des canaux par rapport à la coupure C1
    - Événements de la frontière de la coupure : e13 pour P1 et e33 pour P3
    - État du canal < 1, 3 > = { m3 }
      - m3 a été envoyé en e12 et ne sera reçu qu'en e34
    - État du canal < 3, 1 > = { m4 }
      - m4 a été envoyé en e33 et ne sera reçu qu'en e14

Algorithme de Chandy & Lamport

- Fonctionnement algorithme Chandy & Lamport
  - Un processus Pk est initiateur du lancement de l'algo
    - Il enregistre son état local
    - Il envoie un message marqueur à tous les processus
    - Il arrête son fonctionnement normal tant que l'algo n'est pas fini
  - Un processus Pj, à la réception du marqueur sur son canal < k, j >
    - Enregistre son état local (données, variables ...)
    - Positionne l'état de chacun de ses canaux entrants < i, j > à vide
    - Envoie un marqueur sur tous ses canaux sortants : à tous ses voisins
    - Ces 3 étapes sont exécutées en une séquence atomique
    - Il arrête son fonctionnement normal tant que l'algo n'est pas fini
  - Pour un processus Pj, à la réception du marqueur venant de Pi, sur le canal entrant < i, j >
    - Enregistre l'état du canal < i, j > : tous les messages reçus sur < i, j > depuis la réception du premier marqueur venant de Pk

Algorithme de Chandy & Lamport

- Fonctionnement algorithme Chandy & Lamport (fin)
  - Pour un processus Pj, l'algorithme est fini quand il a reçu un marqueur sur chacun de ses canaux
  - L'état enregistré pour Pj est composé de
    - Son état local (variable, données ...)
    - Les états de tous ses canaux entrants
- Pour constituer l'état global
  - On collecte l'ensemble des états enregistrés par les processus
  - Une fois que l'on sait que tous les processus l'ont enregistré

Algorithme de Chandy & Lamport

- Principe du double marqueur pour savoir quand tout le monde a enregistré son état local
  - Le premier marqueur vient du processus initiateur
    - Le processus Pi « s'arrête » alors (ou passe dans une autre phase de son calcul)
    - Il précise à tous ses voisins qu'il s'est arrêté en leur envoyant un marqueur
    - Il se met en attente de messages l'informant que ses processus voisins se sont arrêtés également
  - A la réception d'un marqueur sur un canal, on sait qu'un de ses voisins s'est arrêté
    - Et que tous ses voisins se sont arrêtés quand on a reçu un marqueur sur tous ses canaux
    - Dans ce cas, l'enregistrement de l'état des canaux est fini, le processus peut reprendre son fonctionnement normal après avoir envoyé son état local et de ses canaux à Pk

Algorithme de Chandy & Lamport

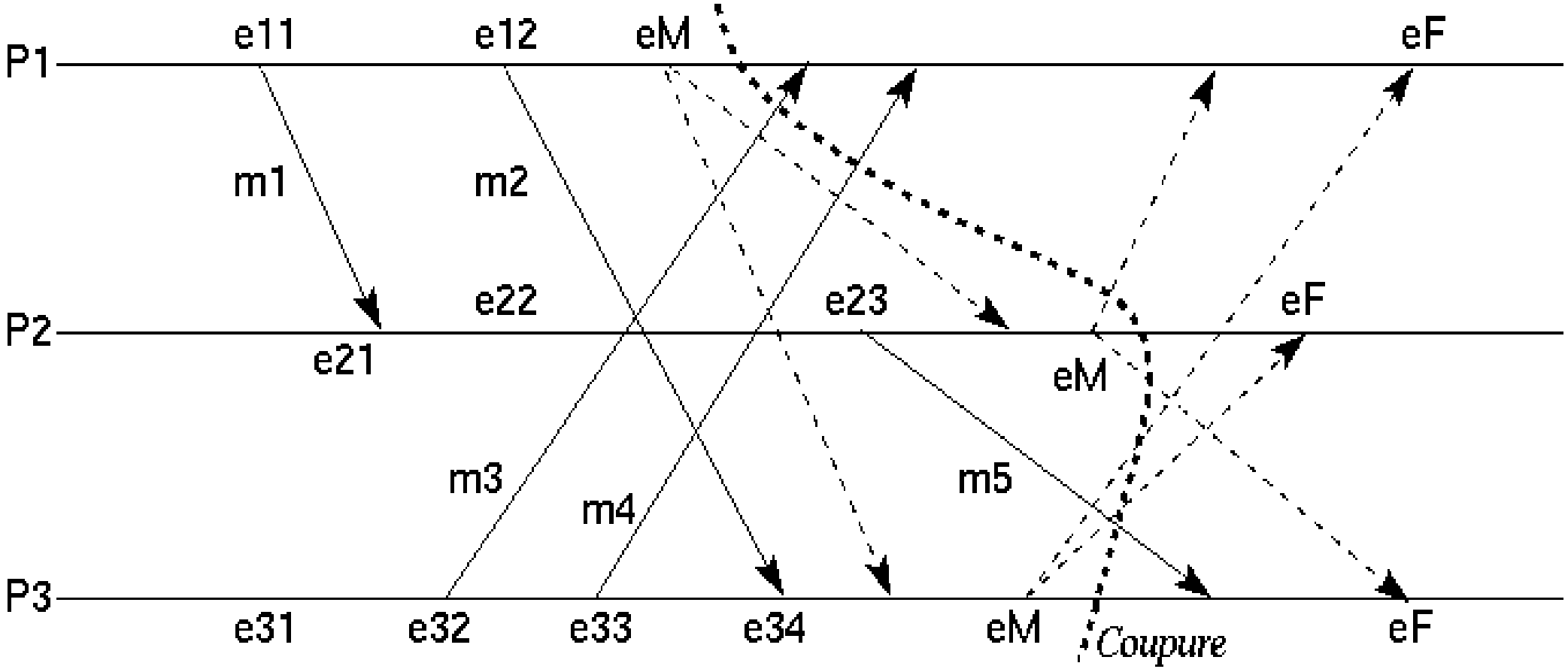
- Intérêt d'enregistrer les messages sur < i, j >
  - Les processus ne sont pas synchronisés et les temps de propagation des messages sont non nuls
  - Ne sait donc pas quand un processus s'arrête et si tous les messages qu'il a envoyé ont été reçus quand il le fait
  - Donc doit enregistrer les messages venant de Pi
    - Comme les canaux sont FIFO, si on reçoit le marqueur de Pi, on sait que tous les messages envoyés par Pi sont maintenus reçus, plus aucun n'est en transit

Algorithme de Chandy & Lamport

- Propriété de l'état global enregistré
  - Correspond à un état cohérent
  - L'algorithme définit une coupure
    - Frontière est formée pour chaque processus par l'événement d'enregistrement de l'état local et de diffusion du marqueur aux autres processus (via une séquence atomique)
  - Cette coupure est cohérente
    - Car canaux FIFO : aucun message ne peut en doubler un autre
      - Localement pour un processus Pj, si un marqueur est reçu sur un canal < i,j >, cela implique qu'aucun message émis par Pi et reçu par Pj avant le marqueur n'a pu être émis après que Pi émette son marqueur
        - Pas de message venant du futur
      - Pour un processus Pj, les événements d'émission des messages en transit (à destination d'autres processus) « coupant » la coupure ont forcément lieu avant son événement local définissant la frontière

Algorithme de Chandy & Lamport

- Exemple avec 3 processus totalement interconnectés



- eM : événement d'enregistrement d'état et diffusion marqueur
- eF : événement où l'état local complet est enregistré
- États canaux : <3,1> = { m3, m4 }, <2,3> = { m5 }, autres sont vides

Reprise d'un système distribué

- A partir d'un état global, on peut relancer un système là où son état a été enregistré
  - On recharge chaque processus avec son état local
  - On réémet les messages qui se trouvaient dans les états des canaux
    - Ils ont été émis mais pas encore reçus par leurs destinataires
    - La réémission d'un message fait partie de l'algorithme de relance, on n'a pas un message d'émission marqué sur le processus
  - Chaque processus reprend son exécution là où son état avait été enregistré
- Besoin d'une coupure cohérente associé à un état
  - Ex. coupure C2 du transparent 39 qui est non cohérente
  - P2 a déjà reçu et traité le message m5
  - P3 redémarre juste après e34 et la première chose que fera P3 est d'envoyer en e35 le message m5 qui sera donc reçu et traité deux fois par P2