Systèmes Distribués — Notes de cours

Yehor KOROTENKO

23 April 2026

Table des matieres

Horloges logiques

§1

Les systèmes distribués posent une difficulté fondamentale qui n'existe pas dans les systèmes centralisés : en l'absence d'une mémoire partagée et d'une horloge physique commune, il est impossible de savoir, en toute généralité, dans quel ordre absolu se sont produits les événements répartis sur plusieurs processus. Ce chapitre construit, pas à pas, les outils conceptuels qui permettent de raisonner sur le temps dans un système distribué : la relation de causalité de Lamport, les horloges scalaires, et les horloges vectorielles.

1.1 Causalité et relation « happens-before »

Dans un système distribué, les processus n'ont accès à aucune horloge globale. La seule information temporelle fiable qu'un processus P_i possède est l'ordre dans lequel **il** a lui-même exécuté ses propres événements. Pour comparer des événements appartenant à des processus différents, il faut donc s'appuyer sur une notion plus abstraite : la **causalité**, c'est-à-dire la possibilité qu'un événement ait pu influencer un autre.

Definition 1.1 (Système distribué) – *Un **système distribué** est un ensemble de n processus $P_1, P_2, ..., P_n$ qui communiquent exclusivement par échange de messages (il n'y a pas de mémoire partagée). Chaque processus P_i possède une **histoire locale** h_i, c'est-à-dire une séquence totalement ordonnée d'**événements** : des événements locaux (calculs internes), des envois de messages (*send*) et des réceptions de messages (*receive*).*

L'ensemble des événements de tous les processus forme l'**histoire globale** du système. La question centrale est : peut-on définir un ordre partiel sur ces événements qui reflète fidèlement les relations de cause à effet ? C'est précisément ce qu'a proposé Leslie Lamport en 1978 avec la relation « *happens-before* ».

Definition 1.2 (Relation happens-before ($\rightarrow$)) – *La relation **happens-before**, notée $\rightarrow$, est le plus petit ordre strict sur les événements du système vérifiant les trois règles suivantes :*

1. ***Ordre local.** Si e et f sont deux événements d'un même processus P_i et que e précède f dans l'histoire locale de P_i, alors $e \rightarrow f$.*
2. ***Communication.** Si e est l'envoi d'un message m et f est la réception de ce même message m (par un processus éventuellement différent), alors $e \rightarrow f$.*
3. ***Transitivité.** Si $e \rightarrow f$ et $f \rightarrow g$, alors $e \rightarrow g$.*

Remarque 1.3 – *La relation $\rightarrow$ définit un **ordre strict partiel** sur l'ensemble des événements : elle est **irréflexive** (aucun événement ne précède lui-même), **asymétrique** (si $e \rightarrow f$ alors il est*

*impossible que $f \to e$) et **transitive** par construction. Elle ne définit pas un ordre total : deux événements peuvent ne pas être comparables.* ◇

Lorsque deux événements ne sont reliés ni dans un sens ni dans l'autre par la relation $\to$, on dit qu'ils sont **concurrents**. Cette notion est fondamentale : elle exprime qu'aucune information ne peut avoir circulé de l'un vers l'autre, et donc que ces deux événements ont pu se produire « simultanément » du point de vue du système.

Definition 1.4 (Événements concurrents) – *Deux événements e et f sont dits **concurrents**, noté $e \parallel f$, si et seulement si*

$$\neg(e \to f) \quad \text{et} \quad \neg(f \to e).$$

Exemple 1.5 (Chaîne causale et concurrence) – *Considérons trois processus P_1, P_2, P_3. P_1 produit un événement e_1, puis envoie un message à P_2. La réception de ce message par P_2 constitue l'événement e_2. P_2 envoie ensuite un message à P_3, reçu par P_3 comme événement e_3. On a donc la **chaîne causale***

$$e_1 \to e_2 \to e_3.$$

*Par transitivité, $e_1 \to e_3$. Séparément, P_1 produit un événement e_4 postérieur à e_1 dans son histoire locale, mais **aucun message** ne relie e_4 à e_2 ou à e_3. On a donc $e_4 \parallel e_2$ et $e_4 \parallel e_3$: ni e_4 n'a pu influencer e_2, ni e_2 n'a pu influencer e_4.* ◇

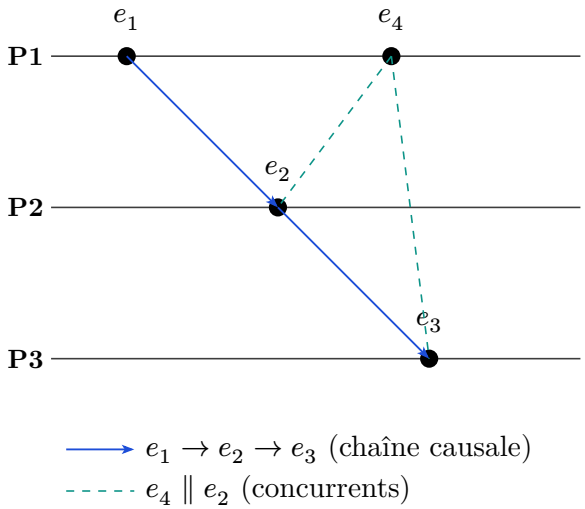


Fig. 1. – Diagramme espace-temps illustrant la causalité. Les flèches bleues représentent la chaîne causale $e_1 \to e_2 \to e_3$; la ligne pointillée verte indique la concurrence $e_4 \parallel e_2$.

1.2 Horloges de Lamport (scalaires)

Disposer d'une relation de causalité est utile sur le plan conceptuel, mais les algorithmes distribués ont souvent besoin de **timestamps** — des entiers attachés aux événements — qui respectent la causalité et permettent des comparaisons simples. Lamport a proposé à cet effet une construction remarquablement simple : l'**horloge logique scalaire**.

L'idée directrice est la suivante. Chaque processus maintient un compteur entier qui représente sa vision locale du « temps logique ». Avant tout événement, ce compteur est incrémenté ; lors d'une réception, le compteur est mis à jour pour tenir compte de l'estampille reçue. Ainsi, si un message porte l'estampille t, le processus récepteur sait qu'au moins t événements ont eu lieu causalement avant cet instant.

Algorithme 1.6 (Horloge de Lamport)**.** Chaque processus P_i maintient un compteur entier C_i, initialisé à 0. Les règles de mise à jour sont :

1. **Avant tout événement local ou envoi :** $C_i := C_i + 1$.
2. **Envoi d'un message :** P_i attache la valeur courante C_i au message sous forme d'estampille ts.
3. **Réception d'un message portant l'estampille** ts : $C_i := \max(C_i, \text{ts}) + 1$.

La règle de réception garantit que le compteur du processus récepteur dépasse strictement l'estampille du message reçu, ce qui encode le fait que la réception est postérieure à l'envoi.

Theoreme 1.7 (Cohérence des horloges de Lamport) – *Pour tout couple d'événements e et f,*

$$e \to f \;\Rightarrow\; L(e) < L(f),$$

où $L(e)$ désigne l'estampille de Lamport de l'événement e.

Preuve. On raisonne par induction sur la définition de $\to$.

- **Règle 1 (ordre local).** Si e et f sont deux événements successifs de P_i avec e avant f, alors C_i a été incrémenté au moins une fois entre e et f, donc $L(f) > L(e)$.
- **Règle 2 (communication).** Si e est l'envoi et f la réception du même message, alors $L(f) = \max(L(f'), L(e)) + 1 \geq L(e) + 1 > L(e)$, où $L(f')$ est la valeur du compteur avant la mise à jour.
- **Règle 3 (transitivité).** Si $e \to g \to f$, par hypothèse d'induction $L(e) < L(g)$ et $L(g) < L(f)$, donc $L(e) < L(f)$.

□

Remarque 1.8 (Limitation fondamentale) – *La réciproque du théorème précédent est **fausse** en général :*

$$L(e) < L(f) \;\not\Rightarrow\; e \to f.$$

*Deux événements concurrents $e \parallel f$ peuvent très bien recevoir des estampilles telles que $L(e) < L(f)$. Les horloges de Lamport permettent de savoir qu'un événement est **peut-être** antérieur à un autre, mais elles ne permettent pas de détecter la concurrence.* ◇

Intuition 1.9 – *Les horloges de Lamport produisent une **extension linéaire** de l'ordre partiel causal : elles placent tous les événements sur une droite numérique cohérente avec $\to$, mais en « aplatissant » des événements concurrents qui n'avaient aucun lien. C'est suffisant pour certains*

algorithmes (exclusion mutuelle de Lamport, par exemple), mais insuffisant pour détecter la concurrence ou caractériser exactement la causalité. ◇

Exemple 1.10 (Trace d'horloge de Lamport) – *Considérons deux processus P_1 et P_2 (compteurs initiaux $C_1 = C_2 = 0$).*

- *P_1 produit un événement local : $C_1 := 1$, estampille $L = 1$.*
- *P_1 envoie un message : $C_1 := 2$, estampille attachée $\mathrm{ts} = 2$.*
- *P_1 produit un autre événement local : $C_1 := 3$, estampille $L = 3$.*
- *P_2 reçoit le message de P_1 ($\mathrm{ts} = 2$) : $C_2 := \max(0, 2) + 1 = 3$, estampille $L = 3$.*
- *P_2 produit un événement local : $C_2 := 4$, estampille $L = 4$.*

*On constate que l'événement local de P_1 (estampille 3) et la réception par P_2 (estampille 3) sont **concurrents**, pourtant ils partagent la même valeur d'horloge. La distinction devient impossible avec les seules horloges scalaires.* ◇

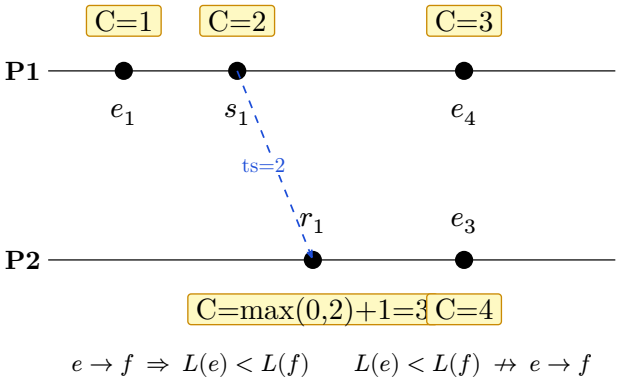


Fig. 2. – Trace d'horloge de Lamport sur deux processus. Les valeurs C encadrées indiquent l'estampille associée à chaque événement ; la flèche pointillée bleue représente le message.

1.3 Horloges vectorielles (Fidge–Mattern)

Les horloges de Lamport souffrent d'une asymétrie gênante : elles préservent la causalité dans un sens ($e \to f \Rightarrow L(e) < L(f)$) mais ne la caractérisent pas dans l'autre. Pour pallier cette limitation, Colin Fidge et Friedemann Mattern ont proposé indépendamment, en 1988, les **horloges vectorielles**, qui réalisent une équivalence complète entre ordre causal et comparaison d'estampilles.

L'idée clé est que chaque processus ne maintient plus un simple entier, mais un **vecteur** d'entiers de taille n — un par processus. La k-ième composante du vecteur de P_i représente le nombre d'événements de P_k dont P_i a connaissance (directement ou par transitvité des messages reçus).

Definition 1.11 (Horloge vectorielle) – *Chaque processus P_i ($1 \le i \le n$) maintient un vecteur $V_i \in \mathbb{N}^n$, initialisé à $(0, 0, ..., 0)$. La composante $V_i[k]$ représente le nombre d'événements du processus P_k que P_i connaît causalement.*

*L'**estampille vectorielle** d'un événement e exécuté par P_i est la valeur V_i **après** la mise à jour liée à e, notée $V(e)$.*

Algorithme 1.12 (Algorithme des horloges vectorielles)**.** Chaque processus P_i maintient un vecteur $V_i \in \mathbb{N}^n$, initialisé à zéro.

1. **Avant tout événement local ou envoi :** $V_i[i] := V_i[i] + 1$.
2. **Envoi d'un message :** P_i attache le vecteur courant V_i au message.
3. **Réception d'un message de P_j portant le vecteur V_{msg} :** pour tout $k \in \{1, ..., n\}$, mettre à jour

$$V_i[k] := \max\big(V_i[k], V_{\text{msg}}[k]\big),$$

puis incrémenter $V_i[i] := V_i[i] + 1$.

La mise à jour par maximum lors d'une réception est cruciale : elle propage la connaissance causale du processus émetteur vers le processus récepteur. Après la réception, P_i connaît tous les événements que P_j connaissait au moment de l'envoi, ainsi que l'envoi lui-même.

Definition 1.13 (Ordre componentwise) – *Soient $V, W \in \mathbb{N}^n$ deux vecteurs d'estampilles. On définit :*

- *$V \leq W$ si et seulement si $V[k] \leq W[k]$ pour tout $k \in \{1, ..., n\}$.*
- *$V < W$ si et seulement si $V \leq W$ et $V \neq W$.*
- *V et W sont **incomparables** si ni $V \leq W$ ni $W \leq V$.*

Theoreme 1.14 (Caractérisation exacte de la causalité) – *Pour tout couple d'événements e et f,*

$$e \to f \;\leftrightarrow\; V(e) < V(f).$$

*Autrement dit, les horloges vectorielles caractérisent **exactement** la relation happens-before.*

Ce résultat est le point fort des horloges vectorielles : contrairement aux horloges de Lamport, la comparaison des estampilles est une condition **nécessaire et suffisante** pour la causalité. On peut ainsi décider algorithmiquement si deux événements sont dans une relation causale ou s'ils sont concurrents, sans ambiguïté.

Corollaire 1.15 – *Deux événements e et f sont concurrents si et seulement si leurs estampilles vectorielles sont incomparables :*

$$e \parallel f \;\leftrightarrow\; V(e) \text{ et } V(f) \text{ sont incomparables.}$$

◇

Remarque 1.16 – *Le coût des horloges vectorielles est proportionnel au nombre de processus n : chaque processus stocke un vecteur de taille n, et chaque message transporte également un tel vecteur. Dans les grands systèmes distribués (n de l'ordre de plusieurs milliers), ce surcoût peut devenir prohibitif. Des variantes compressées (horloges matricielles, plumb-bob clocks, etc.) ont été proposées pour réduire ce coût au prix de garanties affaiblies.* ◇

Exemple 1.17 (Trace vectorielle sur trois processus) – *Considérons trois processus P_1, P_2, P_3 (vecteurs initiaux $[0,0,0]$). Le déroulement suivant illustre le mécanisme :*

1. *P_1 produit e_1 : $V_1 = [1,0,0]$.*
2. *P_1 envoie un message à P_2 (événement s_1) : $V_1 = [2,0,0]$, vecteur $[2,0,0]$ attaché au message.*
3. *P_2 reçoit le message (r_1) : $V_2 := \max([0,0,0],[2,0,0]) = [2,0,0]$, puis $V_2[2]{+}{+} \to V_2 = [2,1,0]$.*
4. *P_2 envoie un message à P_3 (s_2, avec $V_2[2]{+}{+}$) : $V_2 = [2,2,0]$, vecteur $[2,2,0]$ attaché.*
5. *P_3 reçoit le message (r_2) : $V_3 := \max([0,0,0],[2,2,0]) = [2,2,0]$, puis $V_3[3]{+}{+} \to V_3 = [2,2,1]$.*
6. *P_1 produit un événement indépendant e_2 (pas de message) : $V_1 = [3,0,0]$.*

Vérification :

- *$V(e_1) = [1,0,0] < [2,2,1] = V(r_2)$ (componentwise) $\Rightarrow e_1 \to r_2$.*
- *$V(e_2) = [3,0,0]$ et $V(r_1) = [2,1,0]$: $3 > 2$ mais $0 < 1$, donc les vecteurs sont **incomparables** $\Rightarrow e_2 \parallel r_1$.*

◇

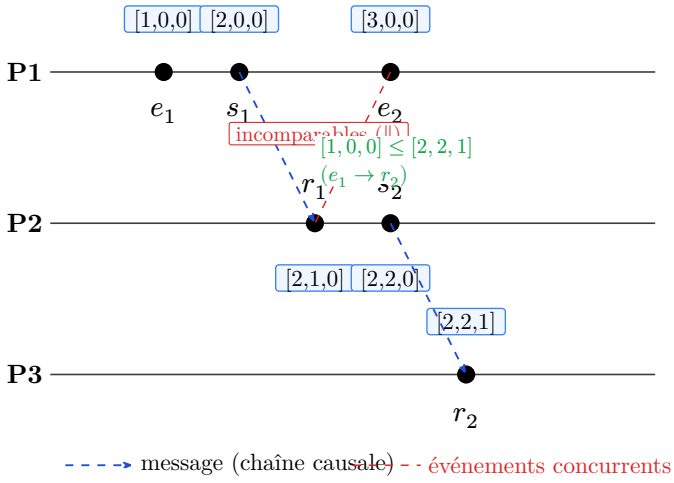


Fig. 3. – Trace d'horloge vectorielle sur trois processus. Les boîtes bleues affichent le vecteur V après chaque événement. La flèche pointillée rouge indique deux événements incomparables (concurrents) ; la note verte montre une relation $\leq$ attestant la causalité.

Algorithmes de diffusion

§2

Dans un système distribué, les nœuds ne partagent pas de mémoire commune et ne peuvent communiquer qu'en échangeant des messages via les canaux du réseau. L'une des opérations fondamentales est la **diffusion** (*broadcast*) : un nœud initiateur souhaite transmettre une information à l'ensemble des nœuds du réseau. Ce chapitre étudie plusieurs algorithmes permettant d'accomplir cette tâche, en analysant leur correction, leur complexité en messages et en temps, ainsi que la question de la **détection de terminaison** globale.

On suppose tout au long de ce chapitre que le réseau est modélisé par un graphe connexe non orienté $G = (V, E)$, où $|V| = N$ désigne le nombre de nœuds et $|E|$ le nombre d'arêtes. Les canaux de communication sont supposés fiables (pas de perte de message) et FIFO.

2.1 Diffusion de base : inondation

2.1.1 Formulation du problème

Avant de présenter l'algorithme, précisons le problème que nous cherchons à résoudre.

Definition 2.1 (Problème de diffusion (broadcast)) – *Étant donné un réseau $G = (V, E)$ connexe et un nœud initiateur $s \in V$ qui détient une valeur m, le **problème de diffusion** consiste à concevoir un algorithme distribué garantissant que tout nœud $v \in V$ reçoit finalement la valeur m, même si les nœuds n'ont aucune connaissance préalable de la topologie du réseau.*

2.1.2 L'algorithme d'inondation

L'idée centrale de l'inondation (*flooding*) est d'une simplicité remarquable : chaque nœud, dès la première réception du message, le retransmet à tous ses voisins. Les messages en double (reçus par un nœud déjà informé) sont silencieusement ignorés.

Algorithme 2.2 (Inondation (Flooding)). **Initialisation.** Chaque nœud $v \in V$ maintient un booléen informé(v), initialisé à faux pour tout nœud sauf l'initiateur.

À l'initiateur s :

1. Marquer informé(s) ← vrai.
2. Envoyer le message m à tous les voisins de s : pour tout $w \in N(s)$, envoyer $\langle m \rangle$ sur le canal (s, w).

À tout nœud $v \neq s$, à la réception d'un message $\langle m \rangle$ depuis un voisin u :

- Si informé(v) = vrai : ignorer le message (doublon).
- Si informé(v) = faux :
  1. Marquer informé(v) ← vrai.
  2. Pour tout voisin $w \in N(v) \setminus \{u\}$: envoyer $\langle m \rangle$ sur le canal (v, w).

Proposition 2.3 (Correction de l'inondation) – *Si le graphe G est connexe, alors l'algorithme d'inondation garantit que tout nœud $v \in V$ reçoit le message m en temps fini.*

Preuve. *Puisque G est connexe, il existe un chemin $s = v_0, v_1, ..., v_k = v$ entre l'initiateur et tout nœud v. On montre par induction sur i que v_i reçoit le message. L'initiateur $v_0 = s$ envoie m à tous ses voisins, en particulier à v_1 ; v_1 reçoit donc m et, étant alors non informé, le retransmet à ses voisins, dont v_2 ; et ainsi de suite. Le nœud $v_k = v$ reçoit ainsi m à l'étape k.* □

Proposition 2.4 (Complexité de l'inondation) – *L'algorithme d'inondation satisfait les bornes de complexité suivantes :*

- ***Messages*** *: au plus $2|E|$ messages sont échangés au total. En pratique, chaque arête (u, v) peut porter au plus un message dans chaque direction : lorsque u informe v et que v rétransmet à u, ce dernier message est simplement ignoré. Donc le nombre de messages est exactement le nombre d'arêtes du réseau couvertes par la diffusion, multiplié par 2, soit $O(|E|)$.*
- ***Temps*** *: la diffusion se termine en $O(\mathrm{diam}(G))$ unités de temps, où $\mathrm{diam}(G)$ est le diamètre du graphe (longueur maximale d'un plus court chemin entre deux nœuds). En effet, chaque nœud à distance d de l'initiateur reçoit le message au plus d tours après le départ.*

Remarque 2.5 – *La règle d'ignorer les messages dupliqués est essentielle pour la terminaison. Sans elle, un nœud pourrait retransmettre indéfiniment des messages reçus par des chemins différents, créant des boucles de diffusion. Le booléen $\mathrm{informé}(v)$ joue exactement le rôle de garde-fou contre cette situation.* ◇

2.1.3 Exemple : réseau à 4 nœuds

Exemple 2.6 (Trace d'inondation sur 4 nœuds) – *Considérons le graphe G avec $V = \{A, B, C, D\}$ et les arêtes $E = \{AB, AC, BC, BD, CD\}$. L'initiateur est A.*

- ***Tour 1.*** *A se marque informé et envoie m à B et C (ses deux voisins).*
- ***Tour 2.*** *B reçoit m de A : se marque informé, retransmet à C et D (en excluant A). C reçoit m de A : se marque informé, retransmet à B et D (en excluant A).*
- ***Tour 3.*** *D reçoit m de B (et/ou de C) : se marque informé, aucune retransmission utile. B reçoit m de C : déjà informé, ignore. C reçoit m de B : déjà informé, ignore.*

Au final, 4 messages utiles sont échangés ($A \to B$, $A \to C$, $B \to D$, $C \to D$) et 2 messages dupliqués ($B \to C$ et $C \to B$) sont ignorés. On a bien $|E| = 5$ arêtes et au plus $2 \times 5 = 10$ messages possibles ; ici seulement 6 sont effectivement envoyés. ◇

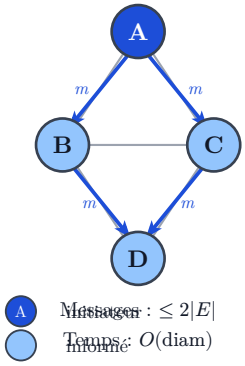


Fig. 4. – Inondation sur un graphe à 4 nœuds. Les nœuds colorés en bleu foncé (initiateur) et bleu clair (informés) montrent l'état final. Les flèches bleues indiquent le flot de messages effectivement transmis.

2.2 Diffusion sur arbre couvrant

2.2.1 Motivation

L'algorithme d'inondation, bien que simple et robuste, souffre d'un défaut : il génère des messages redondants (les doublons ignorés représentent un gaspillage de bande passante). Si le réseau dispose d'un **arbre couvrant** pré-calculé enraciné en l'initiateur, on peut diffuser de manière bien plus économique, sans aucun doublon.

Un arbre couvrant de G est un sous-graphe $T = (V, E_T)$ qui est un arbre (connexe et acyclique) et qui contient tous les nœuds de G. Tout graphe connexe admet au moins un arbre couvrant.

2.2.2 L'algorithme de diffusion sur arbre

Algorithme 2.7 (Diffusion sur arbre couvrant)**. Hypothèse.** Chaque nœud v connaît son père $\text{parent}(v)$ dans l'arbre T (indéfini pour la racine s) ainsi que l'ensemble de ses fils $\text{fils}(v) \subseteq N(v)$.

À la racine s :
1. Envoyer m à chaque fils $f \in \text{fils}(s)$.

À tout nœud interne $v \neq s$, à la réception de m depuis son père :
1. Traiter le message (mémoriser m).
2. Pour tout fils $f \in \text{fils}(v)$: envoyer m à f.

Aux feuilles ℓ (nœuds sans fils), à la réception de m :
1. Traiter le message. Aucune retransmission.

Theoreme 2.8 (Optimalité de la diffusion sur arbre) – *Tout algorithme distribué de diffusion dans un réseau à N nœuds doit envoyer au moins $N-1$ messages. La diffusion sur arbre couvrant est donc optimale en nombre de messages, car elle en envoie exactement $N-1$.*

Preuve. *Considérons l'état initial : seul l'initiateur s détient le message m. Pour qu'un nœud $v \neq s$ reçoive m, il doit recevoir au moins un message le contenant (ou le permettant de le reconstruire). Ces réceptions doivent être* ***distinctes*** *: si on supprime tous les messages*

envoyés, aucun nœud sauf s ne peut être informé. Il faut donc au moins un message par nœud non-initiateur, soit $N-1$ messages au minimum.

La diffusion sur arbre envoie exactement un message par arête de l'arbre, soit $N-1$ messages (un arbre à N nœuds a $N-1$ arêtes). L'algorithme est donc optimal. □

Remarque 2.9 – *La diffusion sur arbre suppose que l'arbre couvrant est **connu à l'avance** par tous les nœuds participants. En pratique, il faut donc d'abord exécuter un algorithme de construction d'arbre couvrant (voir Chapitre 3). Le coût total inclut alors la construction de l'arbre plus la diffusion elle-même. Néanmoins, si l'arbre peut être réutilisé pour plusieurs diffusions successives, l'investissement initial est largement amorti.* ◇

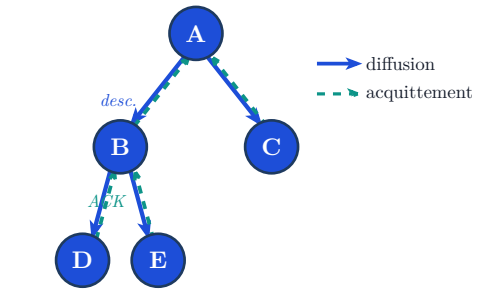


Fig. 5. – Diffusion sur arbre couvrant à 5 nœuds ($N=5$, donc $N-1=4$ messages en descente). Les flèches bleues pleines représentent la phase de diffusion ; les flèches tiretées teal représentent la phase d'acquittement (détaillée à la section suivante).

2.3 Détection de terminaison par vague d'acquittements

2.3.1 Le problème de terminaison

La diffusion sur arbre garantit que tous les nœuds reçoivent le message en exactement $N-1$ messages. Cependant, un problème pratique demeure : **l'initiateur ne sait pas quand la diffusion est terminée**. Il sait qu'il a envoyé ses messages, mais il n'a aucune information sur le moment où le dernier nœud a été informé. Ce problème est particulièrement aigu lorsque la diffusion sert à déclencher une action distribuée et que l'initiateur doit attendre la fin de cette action avant de continuer.

Definition 2.10 (Terminaison globale de la diffusion) – *On dit que la diffusion est **globalement terminée** au moment où le dernier nœud $v \in V$ a reçu et traité le message m. La **détection de terminaison** consiste à permettre à l'initiateur (ou à un observateur désigné) de savoir avec certitude quand cet instant est atteint.*

2.3.2 La vague d'acquittements

La solution classique repose sur une **deuxième phase** qui remonte l'arbre en sens inverse sous forme d'**acquittements** (ACK). L'invariant fondamental est le suivant : un nœud u envoie un

ACK à son père si et seulement si u lui-même a reçu le message **et** tous les nœuds du sous-arbre enraciné en u l'ont également reçu.

Algorithme 2.11 (Vague d'acquittements (ACK wave)). **Phase 1 — Descente (diffusion standard sur arbre).** Exécuter l'algorithme de diffusion sur arbre couvrant décrit à la section précédente.

Phase 2 — Remontée (vague ACK).

Aux feuilles ℓ, immédiatement après avoir reçu m :
1. Envoyer $\langle \text{ACK} \rangle$ au père $\text{parent}(\ell)$.

À tout nœud interne $v \neq s$, après avoir reçu m :
1. Attendre de recevoir un ACK de **chacun** de ses fils : $\forall f \in \text{fils}(v)$, attendre $\langle \text{ACK} \rangle$ de f.
2. Une fois tous les ACKs reçus : envoyer $\langle \text{ACK} \rangle$ à $\text{parent}(v)$.

À la racine s :
1. Attendre de recevoir un ACK de chacun de ses fils.
2. Lorsque tous les ACKs sont reçus : **déclarer la diffusion globalement terminée**.

Theoreme 2.12 (Correction de la vague d'acquittements) – *Lorsque l'initiateur s reçoit un ACK de tous ses fils, la diffusion est effectivement terminée, c'est-à-dire que tout nœud $v \in V$ a reçu et traité le message m.*

Preuve. *On montre par induction structurelle sur l'arbre T que, pour tout nœud u, le nœud u envoie un ACK à son père si et seulement si (a) u a reçu m et (b) tous les nœuds du sous-arbre T_u enraciné en u ont reçu m.*

- ***Base.*** *Pour une feuille ℓ : $T_\ell = \{\ell\}$. La feuille envoie un ACK dès qu'elle reçoit m ; il n'y a pas de sous-arbre non trivial. La propriété est vérifiée.*
- ***Induction.*** *Soit u un nœud interne de fils $f_1, \ldots, f_k$. Par hypothèse d'induction, f_i envoie un ACK à u si et seulement si tout le sous-arbre T_{f_i} a reçu m. Le nœud u attend les ACKs de **tous** ses fils, donc il envoie un ACK à son père si et seulement si u a reçu m ET $T_{f_1}, \ldots, T_{f_k}$ ont tous reçu m, ce qui est exactement T_u tout entier.*

À la racine, $T_s = V$ tout entier. Donc quand s reçoit les ACKs de tous ses fils, V tout entier a reçu m. □

Proposition 2.13 (Complexité de la vague d'acquittements) – *La vague d'acquittements utilise exactement $2(N-1)$ messages au total :*
- *$N-1$ messages lors de la phase de descente (un par arête de l'arbre, descendante).*
- *$N-1$ messages lors de la phase de remontée (un ACK par arête de l'arbre, montante).*

Le temps de terminaison est $O(h)$ où h est la hauteur de l'arbre, car la descente et la remontée parcourent chacune au maximum h niveaux.

Remarque 2.14 – *Ce schéma de diffusion suivie d'une vague d'acquittements est extrêmement général. Il peut être adapté à n'importe quel calcul distribué sur arbre : chaque nœud peut*

effectuer un calcul local et remonter un résultat (par exemple une somme, un minimum, un vote) plutôt qu'un simple ACK. On parle alors de ***réduction distribuée*** *(*distributed reduce*), qui constitue la brique fondamentale de nombreux algorithmes parallèles et distribués.* ◇

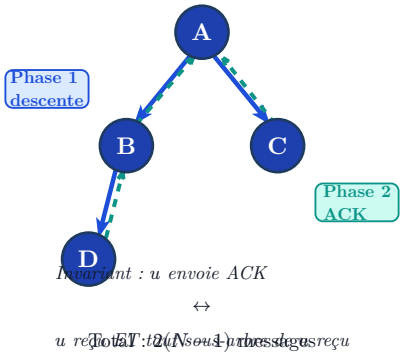


Fig. 6. – Vague d'acquittements sur un arbre à 4 nœuds ($N = 4$). Les flèches bleues pleines indiquent la phase de descente (diffusion) ; les flèches tiretées teal indiquent la remontée des ACKs. Au total : $2(N-1) = 6$ messages.

Construction d'arbres couvrants

§3

Les arbres couvrants occupent une place centrale dans les algorithmes distribués. Comme nous l'avons vu au chapitre précédent, disposer d'un arbre couvrant permet d'effectuer des diffusions optimales en $N-1$ messages et d'organiser des calculs collectifs (réductions, vagues d'acquittements) de manière structurée. Ce chapitre étudie comment construire un tel arbre de façon distribuée, c'est-à-dire sans qu'aucun nœud ne connaisse à l'avance la topologie globale du réseau.

Nous présentons trois familles d'approches aux caractéristiques très différentes :

1. L'**exploration séquentielle** par jeton DFS, qui visite les nœuds un par un avec une garantie de profondeur ;
2. L'**exploration parallèle** par inondation BFS, qui construit un arbre de plus courts chemins très rapidement ;
3. Les **vagues synchronisées** de Bellman-Ford, qui généralisent aux graphes pondérés.

On suppose dans tout ce chapitre que le réseau est modélisé par un graphe connexe non orienté $G = (V, E)$ avec $|V| = N$ et $|E| = M$.

Definition 3.1 (Arbre couvrant) – *Un **arbre couvrant** (*spanning tree*) de $G = (V, E)$ est un sous-graphe $T = (V, E_T)$ tel que :*

- *$E_T \subseteq E$ (les arêtes de T sont des arêtes de G) ;*
- *T est connexe ;*
- *T est acyclique.*

Tout graphe connexe admet au moins un arbre couvrant. Un arbre couvrant à N nœuds possède exactement $N-1$ arêtes.

*Étant donné une racine $s \in V$, on parle d'**arbre couvrant enraciné** : chaque nœud $v \neq s$ possède un unique **parent** $\mathrm{parent}(v)$ sur le chemin de v à s dans T, et les nœuds directement reliés à v en dessous de s sont ses **fils**.*

3.1 Exploration séquentielle : parcours en profondeur avec jeton

3.1.1 Motivation

L'approche la plus intuitive consiste à simuler un **parcours en profondeur** (DFS — *Depth-First Search*) de manière distribuée. Un jeton (*token*) physique circule dans le réseau ; à tout instant, exactement un nœud détient le jeton, ce qui garantit une exploration séquentielle sans conflits. Cette approche est particulièrement utile lorsque l'on doit visiter chaque nœud exactement une fois (par exemple pour compter les nœuds, vérifier une propriété globale, ou construire un identifiant unique pour chaque nœud).

3.1.2 L'algorithme DFS avec jeton

Algorithme 3.2 (DFS distribué avec jeton). **État de chaque nœud v** : un booléen visité(v) (initialement faux) et une liste non_visités(v) $= N(v)$ de voisins non encore explorés depuis v.

À l'initiateur s :

1. Marquer visité(s) $\leftarrow$ vrai.
2. Choisir un voisin $w \in N(s)$ quelconque et envoyer le jeton $\langle \text{jeton}, s \rangle$ à w. Retirer w de non_visités(s).

À tout nœud v, à la réception du jeton $\langle \text{jeton}, \text{père} \rangle$:

- **Si visité(v) = vrai** (le nœud a déjà été visité, c'est un retour arrière depuis un chemin alternatif) :
  - Renvoyer le jeton à père immédiatement (retour arrière, arête non-arbre).
- **Si visité(v) = faux** :
  1. Marquer visité(v) $\leftarrow$ vrai ; enregistrer parent(v) = père.
  2. Si non_visités(v) $\neq \emptyset$: choisir $w \in$ non_visités(v), retirer w, envoyer $\langle \text{jeton}, v \rangle$ à w.
  3. Sinon (tous les voisins explorés depuis v) : envoyer le jeton à parent(v) (retour arrière).

Le jeton revient à s (et non_visités(s) $= \emptyset$) : le DFS est terminé. L'arbre est constitué des arêtes $(u, \text{parent}(u))$.

Theoreme 3.3 (Correction du DFS distribué) – *À la fin de l'algorithme DFS avec jeton, le graphe $T = (V, E_T)$ où $E_T = \{(v, \text{parent}(v)) \mid v \neq s\}$ est un arbre couvrant de G enraciné en s, et il s'agit d'un arbre DFS.*

Preuve (esquisse). *On montre par invariant que le jeton ne visite jamais un nœud déjà intégré dans l'arbre via le même chemin. Chaque nœud v est visité au plus une fois (le booléen* visité(v) *l'empêche d'être intégré deux fois). Comme G est connexe, tous les nœuds sont atteignables depuis s et seront éventuellement visités. L'absence de cycle résulte du fait que les arêtes de l'arbre vont toujours de parent à fils.* □

Proposition 3.4 (Complexité du DFS distribué) – *L'algorithme DFS avec jeton satisfait les bornes suivantes :*

- ***Messages** : exactement $2M$ messages, où $M = |E|$. En effet, chaque arête (u, v) est traversée exactement deux fois par le jeton : une fois dans chaque direction (aller et retour). Les arêtes de l'arbre sont traversées une fois vers l'avant (arête arbre DFS) et une fois en retour arrière ; les arêtes non-arbre sont traversées immédiatement en retour (car le nœud visité renvoie le jeton aussitôt).*
- ***Temps** : $O(M)$ étapes de communication (séquentiel — le jeton parcourt toutes les arêtes une par une).*
- ***Caractère séquentiel** : à tout instant, un seul nœud est actif (celui qui détient le jeton). Cela simplifie certaines propriétés de sécurité mais peut être lent sur de grands graphes.*

Exemple 3.5 (Trace DFS sur 4 nœuds) – *Considérons $V = \{A, B, C, D\}$ avec les arêtes AB, BC, CD, DA, AC, BD (graphe complet K_4). L'initiateur est A.*

- *Étape 1 : A envoie le jeton à B. Arête arbre : $A \to B$.*
- *Étape 2 : B (non visité) enregistre* $\text{parent}(B) = A$*, envoie le jeton à C. Arête arbre : $B \to C$.*
- *Étape 3 : C (non visité) enregistre* $\text{parent}(C) = B$*, envoie le jeton à D. Arête arbre : $C \to D$.*
- *Étape 4 : D (non visité) enregistre* $\text{parent}(D) = C$*. D n'a plus de voisins non visités (tous déjà marqués). Retour arrière : $D \to C$.*
- *Étape 5 : C reçoit le jeton en retour, plus de voisins non visités. Retour arrière : $C \to B$.*
- *Étape 6 : B reçoit le jeton en retour, plus de voisins non visités. Retour arrière : $B \to A$.*
- *Étape 7 : A reçoit le jeton. Plus de voisins non visités. DFS terminé.*

Arbre DFS construit : A— B— C— D (chemin). Total : $2 \times 6 = 12$ messages pour $M = 6$ arêtes de K_4. ◇

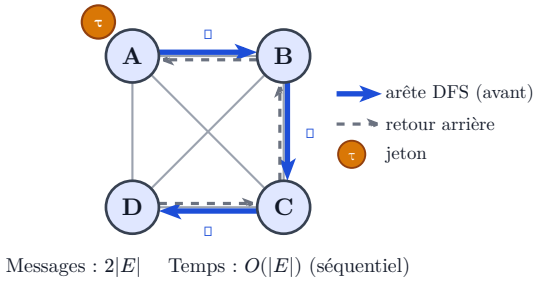


Fig. 7. – DFS distribué avec jeton sur un graphe à 4 nœuds. Les flèches bleues épaisses représentent les arêtes de l'arbre DFS (aller) ; les flèches grises tiretées représentent les retours arrière. Le jeton τ (cercle doré) est montré en position initiale à A.

3.2 Exploration parallèle : BFS par inondation

3.2.1 Motivation

L'algorithme DFS est séquentiel : la progression est lente car le jeton visite les nœuds un à un. Dans de nombreuses applications, on préfère exploiter le **parallélisme inhérent** au réseau pour construire l'arbre couvrant le plus rapidement possible. L'approche par inondation BFS (*Breadth-First Search*) atteint cet objectif : tous les voisins de l'initiateur sont explorés simultanément, puis tous leurs voisins, etc.

3.2.2 L'algorithme BFS par inondation

Algorithme 3.6 (BFS distribué par inondation)**. État de chaque nœud v** : un booléen visité(v) et une référence parent(v) (indéfinie initialement).

À l'initiateur s :

1. Marquer visité(s) $\leftarrow$ vrai.
2. Envoyer $\langle \text{explore}, s \rangle$ à **tous** les voisins de s simultanément.

À tout nœud $v \neq s$, à la réception de $\langle \text{explore}, u \rangle$:

- **Si visité(v) = faux** (premier message reçu) :
  1. Marquer visité(v) $\leftarrow$ vrai ; enregistrer parent(v) $= u$.

2. Envoyer $\langle \text{explore}, v \rangle$ à tous les voisins de v sauf u.

- **Si visité(v) = vrai** : ignorer le message (arête non-arbre, connexion transversale).

Theoreme 3.7 (L'arbre résultant est un arbre BFS) – *À la fin de l'algorithme BFS par inondation, le sous-graphe T des arêtes $(v, \text{parent}(v))$ est un **arbre couvrant BFS** de G enraciné en s : pour tout nœud v, la distance dans T entre s et v est égale à la distance dans G (en nombre de sauts).*

Preuve. *Montrons que le premier message $\langle \text{explore} \rangle$ reçu par v arrive par un voisin u situé à distance $d_{G(s,v)} - 1$ de s, ce qui garantit que $\text{parent}(v)$ est sur un plus court chemin de s à v.*

On procède par récurrence sur $d = d_{G(s,v)}$. Si $d = 1$, v est voisin de s et reçoit le message directement de s au premier tour. Supposons la propriété vraie pour tout nœud à distance $< d$. Un nœud v à distance d est voisin d'au moins un nœud u à distance $d - 1$. Par hypothèse d'induction, u a été visité au tour $d - 1$ et envoie le message au tour d. Aucun nœud plus proche ne peut envoyer de message à v avant le tour d (car ils n'ont été visités qu'à partir du tour $d - 1$). Donc v est visité au tour d par un nœud à distance $d - 1$, ce qui est bien un plus court chemin. □

Proposition 3.8 (Complexité du BFS par inondation) – *L'algorithme BFS par inondation satisfait :*

- ***Messages** : $O(M)$. Chaque arête (u, v) porte au plus deux messages : un dans chaque direction (l'un des deux est toujours ignoré).*
- ***Temps** : $O(\text{diam}(G))$ tours de communication. Chaque nœud à distance d de la racine est visité au tour d, et le nœud le plus éloigné est à distance $\text{diam}(G)$.*

Remarque 3.9 – *Contrairement au DFS, le BFS par inondation est **entièrement parallèle** : plusieurs nœuds peuvent être actifs simultanément. En conséquence, le temps de construction est considérablement réduit — $O(\text{diam})$ au lieu de $O(M)$ — mais le nombre de messages reste similaire. En pratique, sur des réseaux à faible diamètre (par exemple des graphes expanders), le BFS est beaucoup plus rapide que le DFS. En revanche, le BFS ne garantit pas l'ordre de visite des nœuds au sein d'un même niveau, ce qui peut être une limitation pour certaines applications.* ◇

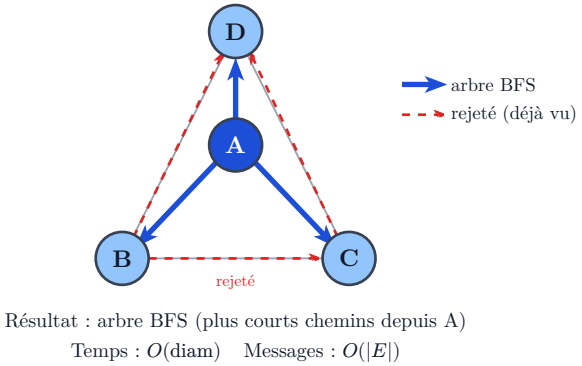


Fig. 8. – BFS par inondation sur un graphe à 4 nœuds. Les flèches bleues épaisses indiquent l'arbre BFS (A est la racine, B, C et D sont ses fils directs). Les flèches rouges tiretées indiquent les messages rejetés (arêtes transversales). Résultat : arbre étoile, reflet de la structure BFS.

3.3 Vagues synchronisées : Bellman-Ford distribué

3.3.1 Motivation

Les deux algorithmes précédents construisent un arbre couvrant quelconque (DFS) ou un arbre de plus courts chemins en nombre de sauts (BFS). Cependant, dans de nombreuses applications réelles, les arêtes du réseau ont des **poids** (latences, débits, coûts) et l'on souhaite construire un **arbre de plus courts chemins pondérés**. C'est l'objectif de l'algorithme de Bellman-Ford distribué.

3.3.2 L'algorithme Bellman-Ford distribué

Soit $G = (V, E, w)$ un graphe connexe pondéré (avec poids $w(u, v) > 0$ pour toute arête (u, v)). On cherche à construire un arbre couvrant enraciné en s où la distance de s à tout nœud v est minimale.

Algorithme 3.10 (Bellman-Ford distribué)**. Initialisation.** Chaque nœud v maintient :

- $d(v)$: estimation courante de la distance de s à v ; initialement $d(s) = 0$ et $d(v) = +\infty$ pour tout $v \neq s$.
- $\text{parent}(v)$: parent courant dans l'arbre ; indéfini initialement.

Chaque tour $k = 1, 2, \ldots$:

1. **Diffusion locale.** Chaque nœud v envoie sa distance courante $d(v)$ à tous ses voisins : pour tout $u \in N(v)$, envoyer $\langle d(v) \rangle$ à u.
2. **Mise à jour.** Chaque nœud u reçoit les distances de tous ses voisins et calcule :

$$d'(u) = \min_{v \in N(u)} \left(d(v) + w(u, v) \right)$$

   Si $d'(u) < d(u)$: mettre à jour $d(u) \leftarrow d'(u)$ et $\text{parent}(u) \leftarrow \arg\min_{v \in N(u)}(d(v) + w(u, v))$.
3. **Condition d'arrêt.** Si aucune distance n'a changé lors de ce tour, l'algorithme converge : l'arbre est constitué des arêtes $(u, \text{parent}(u))$.

Theoreme 3.11 (Convergence de Bellman-Ford distribué) – *L'algorithme Bellman-Ford distribué converge en au plus $N-1$ tours. À la convergence, pour tout nœud v, $d(v)$ est égal à la distance pondérée minimale $\delta(s,v)$ entre s et v dans G.*

Preuve. *On montre par récurrence sur k qu'après k tours, pour tout chemin $s = v_0, v_1, ..., v_k = v$ de longueur k, on a $d(v) \leq w(v_0, v_1) + ... + w(v_{k-1}, v_k)$.*

- ***Base ($k = 0$)*** *: $d(s) = 0 = \delta(s,s)$. Correct.*
- ***Hérédité*** *: Au tour k, le nœud v reçoit $d(u)$ de chaque voisin u. Par hypothèse, $d(u)$ est au plus le poids du plus court chemin de longueur $\leq k-1$ de s à u. Alors $d(u) + w(u,v)$ est au plus le poids d'un chemin de longueur $\leq k$ de s à v via u. En prenant le minimum sur tous les voisins, $d(v)$ atteint le poids du plus court chemin de longueur $\leq k$.*

Tout plus court chemin (sans cycle, puisque les poids sont positifs) comporte au plus $N-1$ arêtes. Donc après $N-1$ tours, toutes les distances sont convergeantes. □

Proposition 3.12 (Complexité de Bellman-Ford distribué) – *L'algorithme Bellman-Ford distribué satisfait :*

- ***Messages*** *: $O(N \cdot M)$. À chaque tour, chaque nœud envoie un message à chacun de ses voisins, soit $2M$ messages par tour. Sur $N-1$ tours : $O(N \cdot M)$ messages au total.*
- ***Temps*** *: $O(N)$ tours de communication.*
- ***Résultat*** *: arbre couvrant de plus courts chemins pondérés depuis s.*

Corollaire 3.13 – *Dans le cas non pondéré (tous les poids égaux à 1), l'algorithme Bellman-Ford distribué produit le même résultat que le BFS par inondation, mais en $O(N)$ tours au lieu de $O(\text{diam})$. Le BFS est donc préférable pour les graphes non pondérés.* ◇

Remarque 3.14 – *L'algorithme de Bellman-Ford distribué est à la base du protocole de routage* ***RIP*** *(*Routing Information Protocol*), l'un des premiers protocoles de routage Internet. Dans ce contexte, les nœuds sont des routeurs, les arêtes sont des liens réseau avec des métriques (nombre de sauts, latences), et chaque routeur calcule sa table de routage en exécutant Bellman-Ford de manière continue. La convergence lente ($O(N)$ tours dans le pire cas) est l'une des limitations connues de RIP, surtout en présence de pannes (le phénomène de « count to infinity »).* ◇

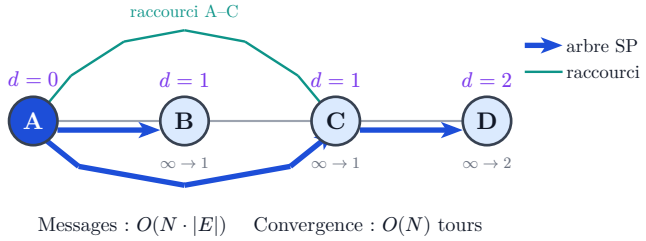


Fig. 9. – Bellman-Ford distribué sur un graphe linéaire à 4 nœuds avec un raccourci A–C. Les étiquettes de distance ($d = 0, 1, 1, 2$) sont calculées après convergence. Les flèches bleues indiquent l'arbre de plus courts chemins résultant ; la courbe teal représente le raccourci qui améliore le chemin vers C.

3.4 Tableau comparatif des algorithmes de construction

Algorithme	Messages	Temps	Résultat	Particularité
DFS avec jeton	$2\|E\|$	$O(\|E\|)$	Arbre DFS	Séquentiel ; visite chaque arête exactement deux fois ; garantit un ordre DFS
BFS par inondation	$O(\|E\|)$	$O(\text{diam})$	Arbre BFS (plus courts chemins en sauts)	Parallèle ; très rapide ; ne gère pas les poids
Bellman-Ford distribué	$O(N \cdot \|E\|)$	$O(N)$	Arbre de plus courts chemins pondérés	Gère les poids ; lent ; base de RIP

Tableau 1. – Comparaison des trois algorithmes distribués de construction d'arbre couvrant. $N = |V|$, $|E|$ = nombre d'arêtes, diam = diamètre du graphe.

En résumé, le choix de l'algorithme dépend du contexte applicatif :

- Si l'on souhaite un arbre de façon séquentielle et garantie, le **DFS avec jeton** est simple et robuste.
- Si la rapidité de construction est prioritaire et le graphe non pondéré, le **BFS par inondation** est optimal en temps.
- Si le réseau est pondéré et que l'on a besoin de plus courts chemins, **Bellman-Ford distribué** est le choix naturel, au prix d'un plus grand nombre de messages.

Exclusion mutuelle distribuée

§4

Dans un système à mémoire partagée, l'exclusion mutuelle s'obtient facilement grâce à des primitives comme les sémaphores ou les verrous matériels. La situation est radicalement différente dans un système distribué : il n'existe aucune mémoire commune, et les processus ne communiquent qu'en s'échangeant des messages. Le problème de l'exclusion mutuelle distribuée consiste à garantir qu'au plus un processus à la fois exécute une **section critique** (SC), c'est-à-dire un fragment de code accédant à une ressource partagée.

Ce chapitre présente cinq approches classiques, du plus simple au plus élaboré. Chacune réalise un compromis entre le nombre de messages échangés par entrée en SC, la robustesse aux pannes et la complexité de mise en œuvre.

Definition 4.1 (Section critique et exclusion mutuelle distribuée) – *Soit n processus $P_1, ..., P_n$ se partageant une ressource. On dit qu'un protocole d'exclusion mutuelle est **correct** s'il satisfait simultanément :*

1. ***Sûreté** : à tout instant, au plus un processus se trouve en section critique.*
2. ***Vivacité** : toute demande d'entrée en SC est éventuellement accordée (absence d'interblocage et d'attente infinie).*
3. ***Équité** : les demandes sont traitées dans un ordre juste, idéalement FIFO selon leurs horodatages.*

La métrique standard pour comparer ces algorithmes est le **nombre de messages échangés par entrée en SC**.

4.1 Solution centralisée

La solution la plus immédiate est d'élire un processus coordinateur qui gère l'accès à la ressource. Lorsque P_i souhaite entrer en SC, il envoie une requête au coordinateur. Celui-ci maintient une file d'attente des demandes en suspens et accorde l'accès selon l'ordre FIFO. Quand P_i a terminé, il notifie le coordinateur par un message de libération.

Cette approche nécessite trois messages par entrée en SC, indépendamment du nombre de processus. Elle est donc extrêmement économique en termes de communication. En revanche, elle crée un **point unique de défaillance** : si le coordinateur tombe en panne, tout le système est bloqué. De plus, le coordinateur devient un goulot d'étranglement à mesure que le nombre de processus croît.

Algorithme 4.2 (Solution centralisée)**. Initialisation** : un processus P_c est désigné coordinateur.

Protocole pour P_i (demandeur) :
1. Envoyer $\mathsf{REQ}(i)$ au coordinateur.

2. Attendre la réception de GRANT.
3. Exécuter la section critique.
4. Envoyer RELEASE(i) au coordinateur.

Protocole pour P_c (coordinateur) :

1. À la réception de REQ(i) : si SC libre, envoyer GRANT à P_i, sinon mettre i en file d'attente.
2. À la réception de RELEASE(i) : retirer P_i de la SC ; s'il y a des demandes en attente, envoyer GRANT au premier de la file.

Proposition 4.3 (Complexité de la solution centralisée) – *La solution centralisée nécessite exactement **3 messages** par entrée en SC :*

- *1 message* REQ *de P_i vers le coordinateur,*
- *1 message* GRANT *du coordinateur vers P_i,*
- *1 message* RELEASE *de P_i vers le coordinateur.*

Remarque 4.4 – *L'algorithme centralisé est correct (sûreté et vivacité garanties par la gestion FIFO de la file). Cependant, son déploiement suppose qu'un coordinateur ait préalablement été élu — ce qui est lui-même un problème non trivial dans un système distribué (voir Chapitre 6). La panne du coordinateur nécessite un mécanisme de recouvrement.* ◇

4.2 Algorithme de Lamport (1978)

L'algorithme de Lamport est la première solution entièrement distribuée à l'exclusion mutuelle. Chaque processus maintient une **file locale** des demandes en cours, ordonnée par les horodatages de Lamport. L'idée centrale est que chaque processus peut calculer localement quel processus a la priorité, à condition d'avoir connaissance de toutes les demandes en circulation.

Pour entrer en SC, P_i diffuse sa demande à tous les processus et attend deux conditions : (a) sa demande est minimale dans toutes les files locales, et (b) il a reçu un accusé de réception de chaque autre processus postérieur à sa propre demande. La sortie est également diffusée afin que tous les processus retirent la demande de leurs files.

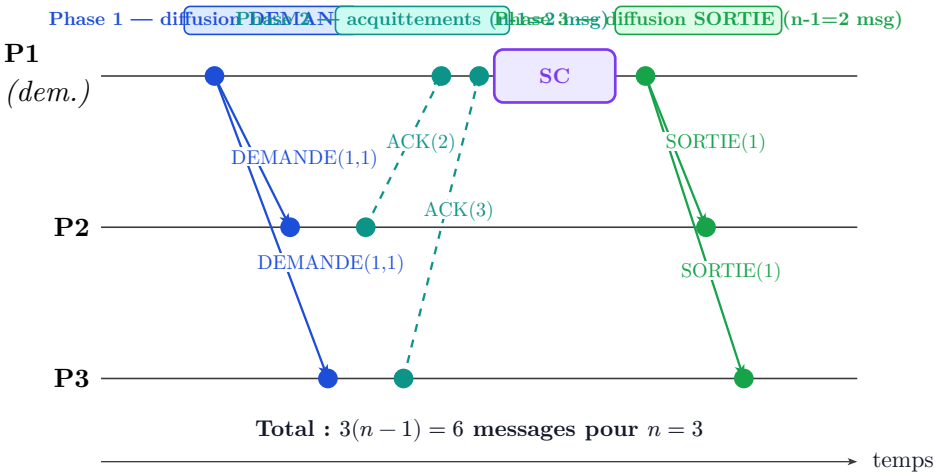


Fig. 10. – Diagramme espace-temps de l'algorithme de Lamport pour $n = 3$. Les trois phases (diffusion de la demande, acquittements, diffusion de la sortie) produisent $3(n-1) = 6$ messages.

Algorithme 4.5 (Algorithme de Lamport). **Structures de données** : chaque P_i maintient une file Q_i et une horloge logique C_i.

Règle 1 — Demande d'entrée : P_i incrémente C_i, diffuse $\mathsf{DEMANDE}(C_i, i)$ à tous les autres processus, et ajoute (C_i, i) à Q_i.

Règle 2 — Réception de $\mathsf{DEMANDE}(t, j)$: P_i met à jour $C_i \leftarrow \max(C_i, t) + 1$, ajoute (t, j) à Q_i, et répond par $\mathsf{ACK}(C_i, i)$ à P_j.

Condition d'entrée en SC pour P_i : (C_i, i) est le minimum de Q_i **et** P_i a reçu un message de chaque P_j $(j \neq i)$ avec horodatage strictement supérieur à C_i.

Règle 3 — Sortie de SC : P_i diffuse $\mathsf{SORTIE}(C_i, i)$ à tous ; à la réception, chaque P_j retire (C_i, i) de Q_j.

Theoreme 4.6 (Correction de l'algorithme de Lamport) – *L'algorithme de Lamport satisfait les propriétés de sûreté, de vivacité et d'équité (ordre FIFO par horodatage).*

Sûreté *: Supposons par l'absurde que P_i et P_j soient simultanément en SC, avec $C_i < C_j$ (ou $C_i = C_j$ et $i < j$). Alors $(C_i, i) <_{\text{lex}} (C_j, j)$. Lorsque P_j entre en SC, il doit avoir reçu un message de P_i postérieur à C_i, donc P_i a déjà reçu l'accusé de réception de P_j — mais cela est impossible si P_j est entré avant d'avoir acquitté P_i.*

Vivacité *: Le bon ordre de la file et la propagation des ACK garantissent qu'aucune demande ne reste bloquée indéfiniment, en l'absence de panne.*

Proposition 4.7 (Complexité de l'algorithme de Lamport) – *L'algorithme de Lamport nécessite $3(n-1)$ messages par entrée en SC :*

- *Phase 1 (diffusion DEMANDE) : $n-1$ messages,*
- *Phase 2 (ACK) : $n-1$ messages,*
- *Phase 3 (diffusion SORTIE) : $n-1$ messages.*

Remarque 4.8 – *La phase de sortie est coûteuse : diffuser* SORTIE *à $n-1$ processus n'est nécessaire que pour maintenir les files cohérentes. Ricart et Agrawala ont observé que cette phase pouvait être éliminée grâce aux* ***réponses différées.*** ◇

4.3 Algorithme de Ricart et Agrawala (1981)

L'algorithme de Ricart et Agrawala optimise celui de Lamport en supprimant la diffusion de sortie. L'idée clé est la suivante : lorsqu'un processus P_i est en SC (ou a une demande prioritaire), il **diffère** sa réponse aux demandes concurrentes. Quand P_i sort de la SC, il envoie directement ses réponses différées — ce qui revient implicitement à notifier les processus concernés sans diffusion globale.

Deux processus P_i et P_j ont des priorités comparables via leurs horodatages : la demande avec le plus petit horodatage (et, à égalité, le plus petit identifiant) est prioritaire. Si P_i reçoit la demande de P_j et que P_i est prioritaire, il diffère sa réponse jusqu'à sa propre sortie de SC. Sinon, il répond immédiatement.

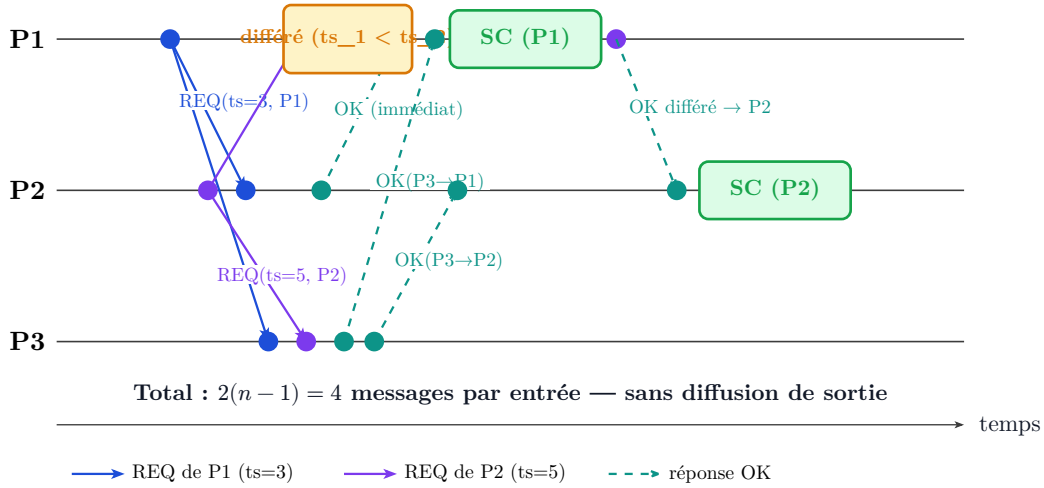


Fig. 11. – Diagramme espace-temps de l'algorithme de Ricart-Agrawala. P1 (ts=3) est prioritaire sur P2 (ts=5). P2 répond immédiatement à P1 ; P1 diffère sa réponse à P2, qu'il envoie en sortant de SC. Total : $2(n-1) = 4$ messages.

Algorithme 4.9 (Algorithme de Ricart-Agrawala)**. Structures de données** : chaque P_i maintient une horloge C_i, un compteur $\mathsf{nb\_ack}_i$ et un ensemble $\mathsf{différes}_i$.

Demande d'entrée de P_i :

1. $C_i \leftarrow C_i + 1$; mémoriser $t_i = C_i$.
2. Diffuser $\mathsf{REQ}(t_i, i)$ à tous les P_j $(j \neq i)$.
3. Attendre $n-1$ réponses OK.
4. Entrer en SC.

Réception de $\mathsf{REQ}(t_j, j)$ par P_i :

- Si P_i **n'est pas** en train de demander la SC, ou si P_i demande la SC mais $(t_j, j) <_{\text{lex}} (t_i, i)$ (i.e. P_j est prioritaire) : répondre immédiatement OK à P_j.
- Sinon (P_i en SC ou prioritaire sur P_j) : ajouter j à $\mathsf{différes}_i$.

Sortie de SC de P_i :

1. Pour chaque $j \in \mathsf{différes}_i$: envoyer OK à P_j.
2. Vider $\mathsf{différes}_i$.

Theoreme 4.10 (Correction de Ricart-Agrawala) – *L'algorithme de Ricart-Agrawala satisfait la sûreté et la vivacité.*

***Sûreté** : Si P_i et P_j demandent la SC simultanément avec $(t_i, i) <_{\text{lex}} (t_j, j)$, alors P_j recevra le REQ de P_i et répondra immédiatement (car P_i est prioritaire), mais P_i ne répondra à P_j qu'après sa sortie de SC. Donc P_j ne pourra pas entrer avant P_i, et a fortiori pas en même temps.*

***Vivacité** : En l'absence de panne, les réponses différées sont toujours finalement envoyées, garantissant la progression.*

Proposition 4.11 (Complexité de Ricart-Agrawala) – *L'algorithme de Ricart-Agrawala nécessite* $2(n-1)$ *messages par entrée en SC :*

- *Phase 1 (diffusion REQ) :* $n-1$ *messages,*
- *Phase 2 (réponses OK, immédiates ou différées) :* $n-1$ *messages.*

Gain par rapport à Lamport : $n-1$ *messages (la phase SORTIE est supprimée).*

4.4 Ricart-Agrawala avec jeton explicite

Les deux algorithmes précédents sont dits **sans jeton** : chaque processus décide localement de son droit d'entrée. Une variante populaire utilise un **jeton** (token) matérialisant explicitement le droit d'accès. Un seul jeton existe dans le système ; le posséder est une condition nécessaire et suffisante pour entrer en SC.

Le jeton est un tableau J de taille n où $J[i]$ enregistre le nombre de fois que P_i est entré en SC depuis la création du jeton. Chaque P_i maintient également un tableau D de **demandes** où $D[i]$ est le numéro de séquence de la dernière demande de P_i.

Algorithme 4.12 (Ricart-Agrawala avec jeton)**. Structures de données** : tableau global $J[1..n]$ (porté par le jeton), tableau local $D_{i[1..n]}$ pour chaque processus.

Demande d'entrée de P_i :

1. Si P_i détient déjà le jeton : entrer directement en SC.
2. Sinon : incrémenter $D_{i[i]}$, diffuser $\mathsf{DEMANDE}\big(D_{i[i]}, i\big)$ à tous.
3. Attendre la réception du jeton.
4. Entrer en SC.

Réception de $\mathsf{DEMANDE}(k, i)$ par P_k (détenteur du jeton) :

- Si P_k n'est pas en SC et $D_{k[i]} > J[i]$: envoyer le jeton à P_i.

Sortie de SC de P_i (détenteur du jeton) :

1. $J[i] \leftarrow J[i] + 1$.
2. Parcourir $j = (i+1) \bmod n, \ldots$ jusqu'à trouver j tel que $D_{i[j]} > J[j]$: envoyer le jeton à P_j.
3. Si aucun j trouvé : conserver le jeton.

Proposition 4.13 (Complexité de R-A avec jeton) – *L'algorithme R-A avec jeton nécessite* ***au plus*** $\mathbf{2(n-1)}$ *messages par entrée en SC :*

- $n-1$ *messages pour la diffusion de la demande,*
- *1 message pour le transfert du jeton (depuis son détenteur actuel),*
- *éventuellement des transferts successifs si le détenteur actuel passe le jeton (au plus* $n-1$ *sauts).*

Dans le meilleur cas (jeton disponible immédiatement), le coût est de $n-1+1$ *messages.*

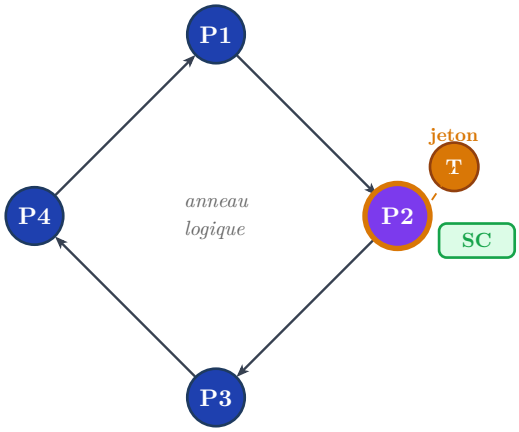
Remarque 4.14 – *L'utilisation d'un jeton apporte une simplification conceptuelle importante : la décision d'entrer en SC est binaire (posséder ou non le jeton). En revanche, la perte du*

jeton (due à une panne du processus le détenant) est catastrophique et nécessite un protocole de régénération spécifique. ◇

4.5 Anneau à jeton

L'anneau à jeton est une solution élégante qui organise les processus en **anneau logique** $P_1 \to P_2 \to \ldots \to P_n \to P_1$. Un unique jeton circule en permanence dans cet anneau dans le sens des aiguilles d'une montre. Lorsqu'un processus reçoit le jeton, il peut l'utiliser pour entrer en SC ; s'il n'en a pas besoin, il le transfère immédiatement au processus suivant.

Cette approche est remarquable par sa simplicité : il n'y a aucune négociation ni aucun calcul de priorité. L'équité est garantie naturellement puisque le jeton fait le tour de tous les processus à chaque cycle. Le coût varie selon la position relative du jeton au moment de la demande.



Meilleur cas : 0 message (jeton déjà là) **Pire cas : $N-1$ messages**

Fig. 12. – Anneau à jeton avec $N = 4$ processus. Le jeton (T) est détenu par P2, qui entre en SC. Les flèches indiquent le sens de circulation unidirectionnel.

Algorithme 4.15 (Anneau à jeton)**. Initialisation** : Le jeton est remis à P_1. Tous les processus connaissent leur successeur dans l'anneau.

Protocole pour P_i recevant le jeton :
1. Si P_i souhaite entrer en SC : entrer en SC, exécuter, puis sortir.
2. Transférer le jeton à $P_{(i \bmod n)+1}$.

Protocole pour P_i ne souhaitant pas la SC :
1. Transférer le jeton immédiatement à $P_{(i \bmod n)+1}$.

Proposition 4.16 (Complexité de l'anneau à jeton) –
- ***Meilleur cas*** *: 0 message par entrée en SC (le jeton arrive exactement au bon moment).*
- ***Pire cas*** *: $n-1$ messages par entrée en SC (le jeton vient d'être transmis au processus suivant, et doit faire presque tout le tour).*
- ***En moyenne*** *: $\frac{n}{2}$ messages par entrée en SC.*

Remarque 4.17 – *L'anneau à jeton génère du trafic en permanence, même si aucun processus ne souhaite la SC. Si k processus utilisent fréquemment la SC, la circulation continue du jeton peut*

être avantageuse ; si les demandes sont rares, ce trafic de fond constitue un gaspillage. La perte du jeton (panne d'un processus en transit) exige un protocole de détection et de régénération. ◇

4.6 Comparaison des algorithmes

Intuition 4.18 (Compromis fondamental) – *Il existe un compromis fondamental entre le nombre de messages par entrée en SC et le degré de distribution du contrôle. La solution centralisée est la moins coûteuse en messages (3 par entrée) mais crée un point de défaillance unique. Les solutions distribuées comme Lamport et Ricart-Agrawala nécessitent $O(n)$ messages mais offrent une meilleure tolérance aux pannes. Le jeton (implicite ou explicite) représente un compromis intermédiaire.* ◇

Algorithme	Messages/entrée	Centralisé ?	Tolérance aux pannes
Centralisée	3	Oui	Faible (SPOF)
Lamport	$3(n-1)$	Non	Moyenne
Ricart-Agrawala	$2(n-1)$	Non	Moyenne
R-A avec jeton	$\leq 2(n-1)$	Non	Perte du jeton
Anneau à jeton	$0...n-1$	Non	Perte du jeton

Tableau 2. – Comparaison des algorithmes d'exclusion mutuelle distribuée.

Remarque 4.19 – *Pour n grand, le coût linéaire en n des algorithmes distribués peut devenir prohibitif. Des solutions avancées basées sur des structures en arbre ou en quorum permettent de réduire ce coût à $O(\sqrt{n})$ ou $O(\log n)$ messages, au prix d'une complexité accrue de mise en œuvre.* ◇

Tolérance aux pannes

§5

Tout système distribué réel est soumis à des défaillances : un processus peut s'arrêter inopinément, un lien réseau peut perdre des paquets, ou un composant peut adopter un comportement erroné. La **tolérance aux pannes** désigne la capacité d'un système à continuer de fonctionner correctement — ou du moins à dégrader gracieusement son service — en présence de telles défaillances.

La question centrale est la suivante : combien de processus défaillants un système distribué peut-il tolérer tout en continuant à garantir ses propriétés ? La réponse dépend crucialement du **modèle de panne** adopté. Ce chapitre présente une hiérarchie de modèles de pannes, de la plus bénigne à la plus sévère, ainsi que les techniques de redondance permettant de les tolérer.

5.1 Types de pannes

Les modèles de pannes forment une **hiérarchie d'inclusion** : les pannes les plus graves englobent les moins graves. Un algorithme conçu pour tolérer des pannes byzantines tolère automatiquement les pannes par omission et par crash.

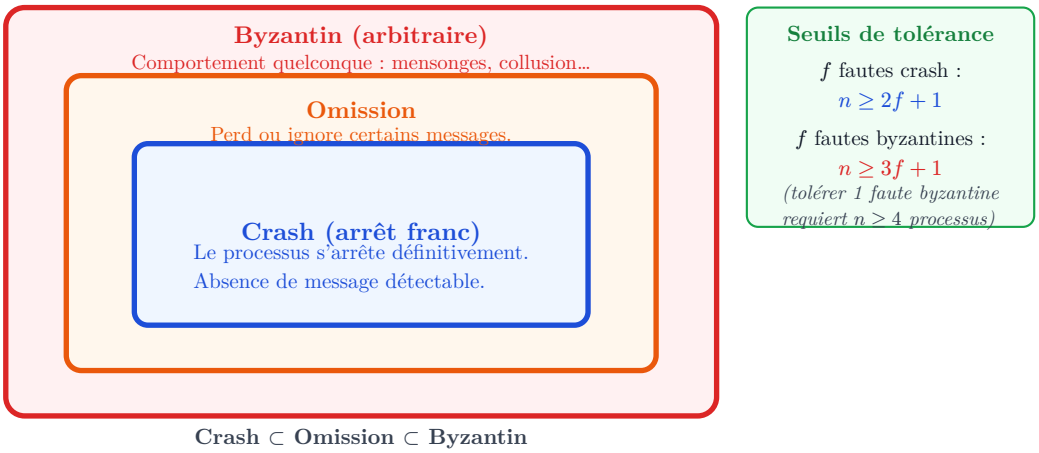


Fig. 13. – Hiérarchie des modèles de pannes. Chaque région englobe la précédente. Les seuils de tolérance indiqués donnent le nombre minimal de processus corrects nécessaires.

5.1.1 Pannes par crash

La panne par crash est le modèle le plus simple et le plus étudié. Un processus qui tombe en panne par crash cesse immédiatement d'envoyer et de recevoir des messages, et ne reprend jamais son exécution (dans le modèle sans recouvrement). L'absence de messages peut être interprétée comme un signal de panne, ce qui rend ce type de défaillance relativement facile à gérer.

Definition 5.1 (Panne par crash (arrêt franc)) – *Un processus P_i subit une **panne par crash** si, à partir d'un certain instant t, P_i n'envoie plus aucun message et ne traite plus aucun message reçu. Les messages envoyés avant t peuvent avoir été ou non reçus par leurs destinataires.*

Dans un système asynchrone pur, il est impossible de distinguer un processus en panne par crash d'un processus simplement lent. C'est pourquoi les détecteurs de pannes (basés sur des timeouts) sont souvent utilisés pour approximer cette distinction, au prix d'une possible erreur.

5.1.2 Pannes par omission

Une panne par omission est plus subtile : le processus continue d'exécuter son code, mais perd parfois des messages à l'émission ou à la réception. Un processus défaillant par omission **semble vivant** mais est peu fiable.

Definition 5.2 (Panne par omission) – *Un processus P_i subit une **panne par omission** s'il omet d'envoyer ou de recevoir certains messages. On distingue :*

- ***Omission à l'émission** : P_i devrait envoyer un message mais ne le fait pas.*
- ***Omission à la réception** : P_i devrait recevoir un message mais ne le traite pas.*
- ***Omission générale** : les deux types peuvent se produire.*

Toute panne par crash est un cas particulier de panne par omission (toutes les émissions et réceptions sont omises à partir d'un certain moment).

Les pannes par omission modélisent notamment les pertes de paquets dans des réseaux non fiables, les tampons d'émission saturés, ou les problèmes transitoires de connectivité.

5.1.3 Pannes byzantines

Les pannes byzantines, introduites par Lamport, Shostak et Pease, représentent le cas le plus général et le plus difficile. Un processus byzantin peut se comporter de manière totalement arbitraire : envoyer des valeurs incorrectes, envoyer des valeurs différentes à des destinataires différents, ou se coordonner avec d'autres processus défaillants pour tromper les processus corrects.

Definition 5.3 (Panne byzantine (faute arbitraire)) – *Un processus P_i subit une **panne byzantine** s'il s'écarte arbitrairement de son comportement spécifié. Formellement, un processus byzantin peut :*

- *envoyer des messages avec des valeurs incorrectes,*
- *envoyer des messages différents à différents processus pour la même étape du protocole,*
- *ne pas envoyer de messages qu'il devrait envoyer,*
- *envoyer des messages non prévus par le protocole,*
- *se coordonner avec d'autres processus défaillants.*

Theoreme 5.4 (Hiérarchie des modèles de pannes) – *Les ensembles de comportements de pannes satisfont :*

$$\text{Crash} \subset \text{Omission} \subset \text{Byzantin}$$

Cette inclusion est stricte : il existe des comportements d'omission qui ne sont pas des crashes (processus vivant mais perdant des messages), et des comportements byzantins qui ne sont pas des omissions (envoi de valeurs fausses).

5.1.4 Seuils de tolérance

Le nombre de processus défaillants qu'un système peut tolérer tout en garantissant ses propriétés dépend du modèle de panne et de la propriété souhaitée.

Theoreme 5.5 (Seuils minimaux de tolérance aux pannes) – *Pour un système distribué de n processus devant tolérer f processus défaillants :*

- ***Tolérance aux crashes** : il suffit de $n \geq f + 1$ processus (mais la disponibilité nécessite souvent $n \geq 2f + 1$ pour permettre la prise de décision majoritaire).*
- ***Tolérance aux pannes byzantines** : il est nécessaire et suffisant d'avoir $n \geq 3f + 1$ processus. Avec $n \leq 3f$ processus, il n'existe **aucun** protocole capable de garantir l'accord en présence de f processus byzantins.*

La borne $n \geq 3f + 1$ pour les pannes byzantines est fondamentale. Son intuition est la suivante : un groupe de f processus byzantins peut se faire passer pour f processus corrects tout en envoyant des informations contradictoires aux $n - f$ processus restants. Pour que ces derniers puissent se mettre d'accord malgré tout, il faut que le groupe des corrects soit suffisamment grand ($n - f > 2f$, soit $n > 3f$).

Fait 5.6 (Impossibilité avec n 3f) – *Si $n \leq 3f$, il n'existe pas de protocole d'accord distribué (consensus, broadcast fiable, etc.) tolérant f pannes byzantines. Ce résultat est prouvé par un argument de partition : les f processus byzantins peuvent partitionner les processus corrects en deux groupes qui reçoivent des informations incompatibles, sans que les corrects puissent les distinguer des byzantins.* ◇

5.2 Redondance et réplication

Face aux pannes, la technique fondamentale est la **redondance** : maintenir plusieurs copies (répliques) de l'état et de la logique de traitement, de sorte qu'une panne isolée ne compromette pas l'ensemble du système. Deux grandes stratégies de réplication existent.

5.2.1 Réplication active

Dans la réplication active, **toutes** les répliques traitent chaque requête et maintiennent le même état. Les clients envoient leurs requêtes à toutes les répliques, et une décision de majorité est prise sur les réponses. Cette approche garantit une disponibilité maximale mais génère un trafic important.

Algorithme 5.7 (Réplication active)**. Structure** : n répliques $R_1, ..., R_n$ maintenant le même état.

Traitement d'une requête q du client :
1. Le client diffuse q à toutes les répliques.
2. Chaque réplique R_i exécute q et répond avec son résultat v_i.
3. Le client attend $f + 1$ réponses identiques et retient la valeur majoritaire.

Propriété : si au plus f répliques sont défaillantes (pannes byzantines), le client obtient la bonne réponse dès lors que $n \geq 3f + 1$.

Remarque 5.8 – *La réplication active requiert que toutes les répliques traitent les requêtes dans le **même ordre total**. Garantir cet ordre est lui-même un problème non trivial, résolu par des protocoles de **diffusion atomique** (voir le chapitre sur le consensus).* ◇

5.2.2 Réplication passive (primaire-sauvegarde)

Dans la réplication passive, une seule réplique — le **primaire** — traite activement les requêtes. Les autres répliques (**sauvegardes**) reçoivent périodiquement l'état du primaire (points de reprise ou **checkpoints**). En cas de panne du primaire, une sauvegarde prend le relais.

Algorithme 5.9 (Réplication passive (primaire-sauvegarde))**. Structure** : 1 primaire P et k sauvegardes $S_1, ..., S_k$.

Traitement normal :
1. Le client envoie sa requête au primaire P.
2. P traite la requête, met à jour son état, diffuse un **checkpoint** aux sauvegardes.
3. P répond au client après confirmation des sauvegardes.

Détection de panne du primaire :
1. Les sauvegardes détectent la panne de P (timeout).
2. Élire une nouvelle sauvegarde comme primaire (par un protocole d'élection).
3. Le nouveau primaire reprend à partir du dernier checkpoint.
4. Notifier les clients de la nouvelle adresse du primaire.

Remarque 5.10 – *La réplication passive génère moins de trafic que la réplication active (les sauvegardes ne traitent pas les requêtes). En contrepartie, le temps de reprise après panne peut être non négligeable (durée de l'élection + application des mises à jour manquantes depuis le dernier checkpoint). Elle tolère bien les pannes par crash mais est inadaptée aux pannes byzantines (le primaire peut envoyer des checkpoints falsifiés).* ◇

5.2.3 Synchronisation après recouvrement

Un nœud qui se remet d'une panne doit **resynchroniser** son état avec les nœuds corrects avant de reprendre sa participation normale au protocole. Cette phase de récupération est délicate :

Algorithme 5.11 (Protocole de resynchronisation)**. Nœud P_i en cours de recouvrement** :
1. Annoncer son redémarrage aux autres nœuds.
2. Demander l'état courant à un sous-ensemble de nœuds corrects.
3. Vérifier la cohérence des réponses reçues (vote majoritaire si nécessaire).

4. Appliquer les mises à jour manquantes depuis le dernier état connu.
5. Rejoindre le protocole normal.

Remarque 5.12 – *Le contexte du théorème CAP (Cohérence, Disponibilité, Tolérance aux Partitions) est pertinent ici : dans un système distribué sujet aux partitions réseau, il est impossible de garantir simultanément la cohérence forte et la disponibilité totale. La tolérance aux pannes impose donc des compromis, formalisés dans la pratique par des niveaux de cohérence variés (cohérence finale, lecture de ma propre écriture, etc.).* ◇

5.3 Récapitulatif

Les différents modèles de pannes imposent des contraintes croissantes sur les algorithmes distribués. Tolérer des pannes byzantines est significativement plus coûteux que tolérer des crashes, tant en termes de nombre de processus nécessaires ($3f + 1$ contre $2f + 1$) qu'en termes de complexité des protocoles. En pratique, les systèmes choisissent leur modèle de panne en fonction des menaces réelles : les pannes par crash suffisent dans un datacenter de confiance, tandis que les pannes byzantines sont nécessaires pour des systèmes ouverts ou adversariaux (blockchain, etc.).

Intuition 5.13 (Redondance et disponibilité) – *La redondance est le seul mécanisme universel de tolérance aux pannes. Plus le modèle de panne est sévère, plus la redondance requise est importante. Cependant, la redondance introduit de nouveaux défis : cohérence des répliques, synchronisation après recouvrement, et gestion des partitions réseau. La conception d'un système tolérant aux pannes consiste essentiellement à trouver le bon équilibre entre redondance, performance et complexité.* ◇

Élection de chef

§6

Dans de nombreux algorithmes distribués, il est commode de disposer d'un processus particulier jouant le rôle de **coordinateur** ou de **chef** : il peut centraliser les décisions, initier un protocole, ou servir de point de synchronisation. Or, dans un système distribué symétrique, tous les processus démarrent dans le même état ; aucun ne joue de rôle privilégié a priori.

Le problème de l'**élection de chef** consiste à faire converger un système distribué vers un état dans lequel exactement un processus se déclare élu, et tous les autres processus reconnaissent ce choix. La seule hypothèse permettant de briser la symétrie est l'existence d'**identifiants uniques** pour les processus.

Definition 6.1 (Problème de l'élection de chef) – *Soit un réseau de n processus $P_1, ..., P_n$, chacun possédant un identifiant unique $\mathrm{id}(P_i) \in \mathbb{N}$. Un algorithme d'élection est **correct** s'il garantit, à partir de tout état initial :*

1. ***Terminaison** : en temps fini, tous les processus terminent.*
2. ***Accord** : exactement un processus se trouve dans l'état « élu ».*
3. ***Validité** : le processus élu est celui ayant le plus grand (ou le plus petit) identifiant parmi tous les processus corrects.*

Les deux algorithmes classiques présentés dans ce chapitre s'appliquent à des topologies en **anneau**, où les processus sont disposés en cercle et communiquent avec leurs voisins.

6.1 Algorithme de Chang-Roberts (extinction sélective)

L'algorithme de Chang et Roberts, publié en 1979, s'applique à un **anneau unidirectionnel** : chaque processus ne peut envoyer des messages que dans un sens (par exemple, dans le sens des aiguilles d'une montre). Le principe est celui de l'**extinction sélective** : chaque processus envoie son identifiant dans l'anneau, et un identifiant est **éliminé** dès qu'il rencontre un processus avec un identifiant plus grand. Seul l'identifiant maximal survit et fait le tour complet de l'anneau, désignant son émetteur comme chef.

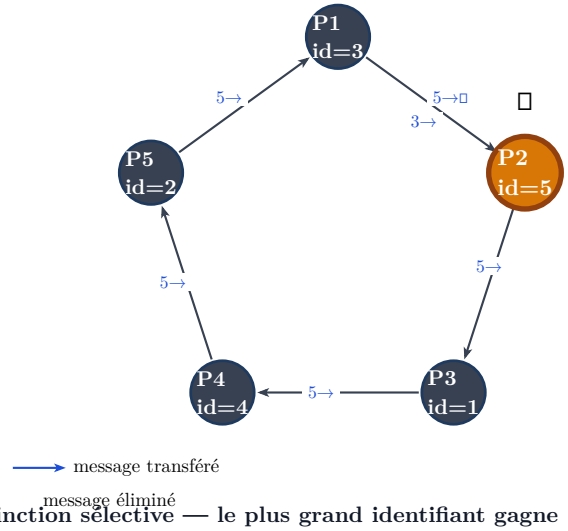


Fig. 14. – Algorithme de Chang-Roberts sur un anneau de 5 processus. L'identifiant id=5 (P2) est transféré par tous les processus et revient à son émetteur, le désignant chef. Les identifiants plus petits sont éliminés dès qu'ils rencontrent un identifiant supérieur.

Algorithme 6.2 (Algorithme de Chang-Roberts)**. Initialisation** : tous les processus sont « actifs ».

Phase d'envoi : chaque processus P_i envoie $\mathrm{id}(P_i)$ à son successeur dans l'anneau.

Règle de traitement : lorsque P_i reçoit un identifiant j :

- Si $j > \mathrm{id}(P_i)$: transférer j au successeur (P_i est dominé, j survit).
- Si $j < \mathrm{id}(P_i)$: supprimer j (message éliminé, P_i est toujours candidat).
- Si $j = \mathrm{id}(P_i)$: P_i se déclare **ÉLU** (son propre message a fait le tour complet).

Diffusion du résultat : le processus élu diffuse un message LEADER(id) à tous les processus pour les informer du résultat.

Theoreme 6.3 (Correction de Chang-Roberts) – *L'algorithme de Chang-Roberts est correct : il termine, élit exactement un processus, et ce processus est celui ayant l'identifiant maximal.*

***Preuve** : Le message portant l'identifiant maximal $M = \max_i \mathrm{id}(P_i)$ ne peut jamais être éliminé (aucun processus n'a un identifiant supérieur à M). Il fait donc le tour complet de l'anneau et revient à $P_{\max}$, qui se déclare élu. Tout autre message $j < M$ est éventuellement éliminé lorsqu'il atteint le processus dont l'identifiant est supérieur à j. Donc exactement un processus — $P_{\max}$ — se déclare élu.* □

6.1.1 Analyse de complexité

La complexité de Chang-Roberts dépend de la disposition des identifiants sur l'anneau.

Proposition 6.4 (Complexité de Chang-Roberts) –

- ***Pire cas** : $\Theta(n^2)$ messages (identifiants disposés en ordre décroissant).*

- ***Cas moyen*** *:* $O(n \log n)$ *messages (pour une permutation aléatoire des identifiants).*

Exemple 6.5 (Pire cas de Chang-Roberts en $\Theta(n^2)$) – *Considérons* n *processus disposés en anneau avec les identifiants dans l'ordre* ***décroissant*** *dans le sens de circulation :* $n, n-1, n-2, ..., 1$.

Numérotons les processus $P_1, P_2, ..., P_n$ *dans le sens de circulation, avec* $\mathrm{id}(P_k) = n+1-k$. *Ainsi* P_1 *a l'identifiant* n *(le maximum),* P_2 *a* $n-1$*, etc.*

Analysons combien de sauts chaque message effectue avant d'être éliminé :

- $\mathrm{id}(P_n) = 1$ *: éliminé immédiatement à* P_1 *(1 saut).*
- $\mathrm{id}(P_{n-1}) = 2$ *: éliminé à* P_1 *après 2 sauts.*
- ...
- $\mathrm{id}(P_2) = n-1$ *: éliminé à* P_1 *après* $n-1$ *sauts.*
- $\mathrm{id}(P_1) = n$ *: fait le tour complet,* n *sauts.*

Total de messages : $1 + 2 + ... + n = \frac{n(n+1)}{2} = \Theta(n^2)$.

Cette configuration est bien le pire cas : dans toute autre configuration, certains messages sont éliminés plus tôt. ◇

Remarque 6.6 – *Le pire cas* $\Theta(n^2)$ *se réalise exactement lorsque les identifiants décroissent dans le sens de circulation. Dans ce cas, chaque message doit « remonter » contre tous les identifiants inférieurs avant d'être arrêté par le maximum. Pour des applications pratiques où* n *est grand, on préférera l'algorithme de Peterson qui garantit* $O(n \log n)$ *dans tous les cas.* ◇

6.2 Algorithme de Peterson (anneau bidirectionnel)

L'algorithme de Peterson, publié en 1982, s'applique à un **anneau bidirectionnel** et garantit une complexité de $O(n \log n)$ messages dans le pire cas. L'idée centrale est une élimination **par phases** : à chaque phase, au moins la moitié des processus encore candidats sont éliminés, ce qui garantit $O(\log n)$ phases, chacune utilisant $O(n)$ messages.

Initialement, tous les processus sont **actifs** (candidats à l'élection). Au cours de chaque phase, chaque processus actif envoie son identifiant courant à ses deux voisins directs et consulte l'identifiant de son voisin gauche et de son « demi-voisin » gauche (le voisin du voisin). Si l'identifiant du voisin direct est le plus grand des trois, ce voisin « survit » et devient le représentant pour la phase suivante ; sinon il est éliminé.

Algorithme 6.7 (Algorithme de Peterson)**. Initialisation** : tous les processus P_i sont actifs, avec identifiant courant $a_i = \mathrm{id}(P_i)$.

Phase répétée jusqu'à élection :

Pour chaque processus actif P_i (simultanément) :

1. Envoyer a_i à droite.
2. Recevoir a_{left} du voisin gauche actif.
3. Envoyer a_{left} à droite (relayer).
4. Recevoir $a_{\text{left,left}}$ (le relais du voisin gauche du voisin gauche).
5. Si $a_{\text{left}} > \max(a_i, a_{\text{left,left}})$: $a_i \leftarrow a_{\text{left}}$ (le voisin gauche survit, devient représentant de P_i).

6. Sinon : P_i est éliminé (passe en mode passif, se contentant de relayer).

Terminaison : lorsqu'un seul processus actif reste, il se déclare élu et diffuse LEADER.

Theoreme 6.8 (Complexité de Peterson en O(n log n)) – *L'algorithme de Peterson nécessite $O(n \log n)$ messages dans le pire cas.*

Preuve (argument de division par deux)** : À chaque phase, considérons les processus actifs. Parmi deux processus actifs consécutifs P_i et son voisin gauche actif P_j, au plus l'un des deux peut « survivre » : P_j survit seulement si son identifiant est strictement supérieur à ceux de ses deux voisins actifs. En particulier, P_i et P_j ne peuvent pas survivre tous les deux simultanément (si P_j survit, c'est que son identifiant est plus grand que celui de P_i, donc P_i ne survit pas, et réciproquement). Ainsi, **au moins la moitié des processus actifs sont éliminés à chaque phase.

Avec n processus actifs initialement, après k phases il reste au plus $\frac{n}{2^k}$ candidats. Le nombre de phases est donc au plus $\lceil \log_2 n \rceil = O(\log n)$.

Chaque phase consomme au plus $2n$ messages (chaque processus actif envoie et reçoit un nombre constant de messages, et les processus passifs relaient). Le total est donc $O(n \log n)$. □

Intuition 6.9 (Intuition de la division par deux) – *L'argument clé de Peterson est que la condition de survie ($a_{\text{left}} > \max(a_i, a_{\text{left,left}})$) est « strictement locale à deux voisins consécutifs » : deux candidats adjacents ne peuvent pas tous deux remplir cette condition simultanément. Cela garantit que le nombre de survivants est au plus la moitié du nombre de candidats, indépendamment de la configuration des identifiants. En $O(\log n)$ phases, on converge nécessairement vers un unique élu.*

Comparer avec Chang-Roberts, où l'absence de mécanisme d'élimination local garantie conduit au pire cas $\Theta(n^2)$: un seul identifiant (le maximum) « balaie » séquentiellement l'anneau, éliminant tous les autres en les rencontrant. ◇

6.2.1 Comparaison des deux algorithmes

Algorithme	Topologie	Pire cas	Cas moyen	Élu
Chang-Roberts	Unidirectionnel	$\Theta(n^2)$	$O(n \log n)$	id max
Peterson	Bidirectionnel	$O(n \log n)$	$O(n \log n)$	id max

Tableau 3. – Comparaison des algorithmes d'élection sur anneau.

Remarque 6.10 – *La bidirectionnalité de l'anneau est essentielle pour Peterson : l'algorithme nécessite de consulter deux voisins dans la même phase, ce qui requiert une communication dans les deux sens. Sur un anneau strictement unidirectionnel, le meilleur algorithme connu a une complexité de $O(n \log n)$ en moyenne mais $\Theta(n^2)$ dans le pire cas (Chang-Roberts). Sur un graphe général, des algorithmes d'élection existent avec une complexité de $O(m + n \log n)$ où m est le nombre d'arêtes.* ◇

Remarque 6.11 – *L'élection de chef est une primitive fondamentale pour de nombreux autres algorithmes distribués. En particulier, la solution centralisée à l'exclusion mutuelle (Chapitre 4) et certains protocoles de consensus (Chapitre 9) supposent l'existence d'un coordinateur préalablement élu. Dans des systèmes dynamiques où les processus peuvent tomber en panne et redémarrer, l'élection doit être périodiquement relancée — c'est le rôle des* ***détecteurs de pannes*** *et des protocoles de* ***ré-élection****.* ◇

Détection de terminaison

§7

Dans un système distribué, la notion même de « fin du calcul » est beaucoup plus subtile que dans un programme séquentiel. Lorsqu'un processus unique termine sa boucle principale, on sait immédiatement que le calcul est achevé. Dans un système distribué, en revanche, plusieurs processus s'exécutent en parallèle et se transmettent des messages : un processus peut devenir inactif, puis être réveillé par un message qu'un autre processus lui avait envoyé avant même de s'endormir. De l'extérieur, il est impossible de distinguer ce cas d'une terminaison effective.

Ce chapitre étudie le problème de la **détection de terminaison** : comment un observateur extérieur — ou l'un des processus lui-même — peut-il déterminer de manière sûre que le calcul distribué a globalement terminé, sans interrompre le calcul ni disposer d'une horloge globale ? Nous présenterons trois approches classiques : le jeton de Dijkstra–Safra pour les anneaux synchrones, l'algorithme de Safra avec compteurs pour les systèmes asynchrones, et le schéma de crédit de Mattern pour les calculs diffusants.

7.1 Le problème de la terminaison

Considérons un ensemble de n processus $P_1, ..., P_n$ qui participent à un calcul distribué. À tout instant, chaque processus est soit **actif** (il calcule ou vient d'envoyer un message), soit **inactif** (il n'effectue aucun calcul local). Un processus inactif peut redevenir actif à tout moment s'il reçoit un message.

L'approche naïve consiste à vérifier périodiquement si tous les processus sont inactifs. Cette vérification est cependant insuffisante, et ce pour une raison fondamentale : un message peut être **en transit** dans le réseau. Si P_1 a envoyé un message à P_2 avant de s'endormir, et si P_2 est également inactif au moment de la vérification, alors l'observateur peut conclure à tort que le calcul est terminé. Mais dès que P_2 recevra le message, il se réveillera et continuera le calcul.

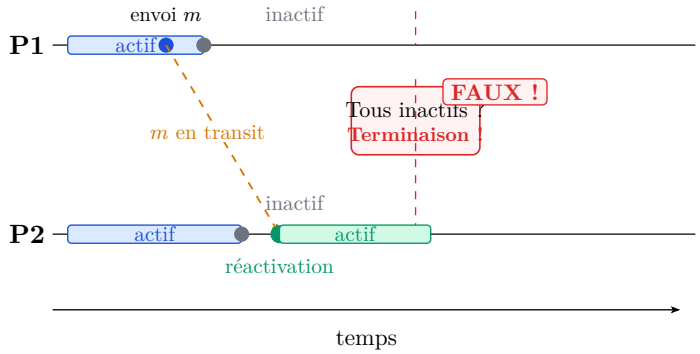
Definition 7.1 (Terminaison globale) – *Un calcul distribué est dit **globalement terminé** si et seulement si les deux conditions suivantes sont simultanément satisfaites :*

1. ***Tous les processus sont inactifs** : pour tout $i \in \{1, ..., n\}$, le processus P_i n'effectue aucune action locale.*
2. ***Aucun message n'est en transit** : pour tout couple (i, j), il n'existe aucun message envoyé par P_i à P_j qui n'ait pas encore été reçu par P_j.*

Cette définition met en lumière la difficulté fondamentale : les messages en transit sont **invisibles** à un observateur externe. Un gestionnaire qui ne surveille que les états des processus ne peut pas savoir combien de messages circulent encore dans le réseau.

Exemple 7.2 (Échec de la détection naïve) – *Supposons deux processus P_1 et P_2. P_1 est actif, envoie un message m à P_2 à l'instant $t = 1.5$, puis devient inactif à $t = 2$. P_2, de son côté, termine ses propres calculs et devient inactif à $t = 2.5$.*

Un gestionnaire qui vérifie les états à $t = 2.5$ constate : P_1 inactif, P_2 inactif. Il déclare la terminaison. C'est une erreur : le message m n'a pas encore été délivré. À $t = 3$, P_2 reçoit m, se réactive, et reprend le calcul. ◇



Attention : processus inactif ≠ terminé — les messages en transit comptent !

Fig. 15. – Illustration de l'échec de la détection naïve. P_1 envoie un message m à $t = 1.5$ puis devient inactif à $t = 2$; P_2 devient inactif à $t = 2.5$. Le gestionnaire déclare à tort la terminaison — mais m est encore en transit et réactivera P_2 à $t = 3$.

Intuition 7.3 – *La difficulté fondamentale est que l'état global visible (les états des processus) et l'état global réel (incluant les messages en transit) divergent. Toute solution au problème doit donc, d'une façon ou d'une autre,* ***comptabiliser les messages en vol****. Les trois algorithmes que nous allons étudier utilisent des mécanismes différents pour réaliser ce comptage : un jeton coloré, des compteurs par processus, ou une fraction de « crédit ».* ◇

7.2 Jeton de Dijkstra–Safra

La première solution que nous présentons est applicable dans un modèle synchrone où les processus sont organisés en **anneau** : $P_1 \to P_2 \to \cdots \to P_n \to P_1$, chaque processus n'ayant pour voisin de droite que le processus suivant. Le processus P_1 joue le rôle d'**initiateur** : c'est lui qui déclenche la détection et interprète le résultat.

L'idée centrale est de faire circuler un **jeton** sur l'anneau. Lorsque le jeton revient à P_1 après un tour complet, P_1 peut tenter de conclure à la terminaison. Pour détecter les messages envoyés « à rebours » dans l'anneau (c'est-à-dire d'un processus P_i vers un processus P_j avec $j < i$, ce qui pourrait réactiver un processus déjà visité par le jeton), on utilise une **couleur** associée au jeton.

Chaque processus est coloré **blanc** (inactif, n'a envoyé aucun message depuis qu'il a tenu le jeton) ou **rouge** (actif, ou a envoyé un message à un processus en amont dans l'anneau depuis la dernière fois qu'il a tenu le jeton). Le jeton lui-même est blanc ou noir.

Algorithme 7.4 (Jeton de Dijkstra–Safra)**.** Chaque processus P_i maintient une couleur locale : **blanc** ou **rouge**.

1. **Initialisation.** P_1 crée un jeton **blanc** et le transmet à P_2.

2. **Réception du jeton par P_i ($i > 1$).**
   - Si P_i est **rouge** (actif) : P_i attend de devenir inactif.
   - Si P_i a envoyé des messages depuis son dernier passage du jeton : le jeton devient **noir** (contamination irréversible).
   - P_i transmet le jeton (possiblement noirci) à P_{i+1}.
3. **Réception du jeton par P_1.**
   - Si le jeton est **blanc** et P_1 a été inactif pendant tout le tour : **terminaison détectée**.
   - Sinon : P_1 détruit le jeton et en recrée un blanc ; recommencer.

Theoreme 7.5 (Correction du jeton de Dijkstra–Safra) – *L'algorithme du jeton de Dijkstra–Safra est **correct** : si P_1 détecte la terminaison (jeton blanc revenant à P_1 avec P_1 inactif pendant tout le tour), alors le calcul est effectivement globalement terminé.*

Réciproquement, si le calcul termine, P_1 détectera la terminaison en un nombre fini de tours.

Preuve. **Sûreté (pas de fausse détection).** Supposons que P_1 reçoive un jeton blanc alors qu'il était inactif pendant tout le tour. Un jeton blanc signifie qu'aucun processus P_i visité n'a envoyé de message à un processus P_j avec $j < i$ (sinon le jeton serait noir). Tous les processus ayant passé le jeton étaient inactifs au moment du passage. Aucun message envoyé à rebours n'a pu réactiver un processus déjà visité. Donc, au moment où P_1 conclut, tous les processus sont bien inactifs et aucun message vers un processus antérieur n'est en transit.

Vivacité (détection éventuelle). Si le calcul termine à un instant T, alors à partir de T tous les processus restent inactifs et n'envoient plus de messages. Le prochain tour de jeton initié après T sera entièrement blanc, et P_1 détectera la terminaison. □

Remarque 7.6 – *Le mécanisme de contamination par la couleur noire est la clef de la correction. Un processus P_i qui envoie un message à P_j avec $j < i$ **après** que le jeton ait déjà visité P_j pourrait faire croire à tort que P_j est définitivement inactif. En noircissant le jeton, P_i force P_1 à lancer un nouveau tour, permettant ainsi à P_j d'être observé à nouveau après réception du message.* ◇

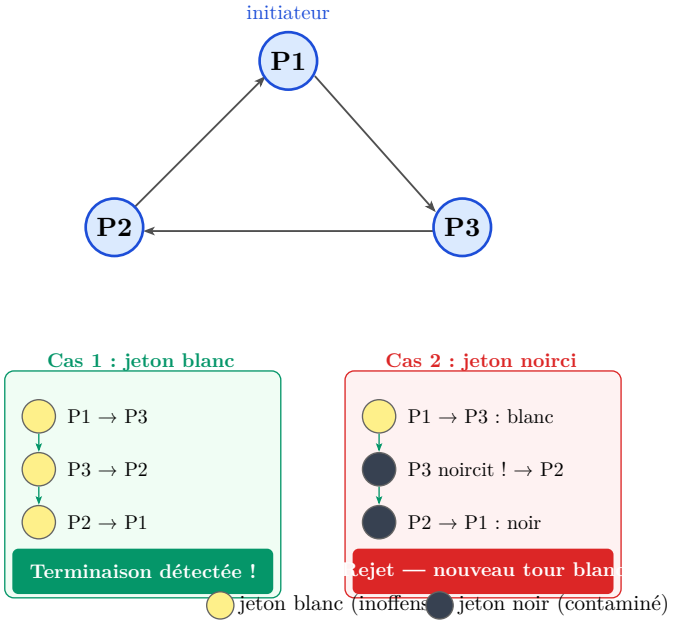


Fig. 16. – Jeton de Dijkstra–Safra sur un anneau de trois processus. **Gauche :** le jeton reste blanc tout au long du tour — tous les processus étaient inactifs et aucun message n'a été envoyé à rebours ; P_1 conclut à la terminaison. **Droite :** un processus contamine le jeton en noir car il a envoyé des messages depuis son dernier passage ; P_1 lance un nouveau tour avec un jeton blanc.

7.3 Algorithme de Safra (asynchrone)

Le jeton de Dijkstra–Safra suppose un modèle synchrone et une topologie en anneau. Dans un système **asynchrone** de topologie quelconque, on ne peut pas garantir que les messages sont délivrés dans un ordre connu. Safra a proposé une généralisation qui remédie à ces limitations en associant à chaque processus un **compteur de messages** qui trace explicitement le nombre de messages envoyés et reçus.

L'idée est la suivante : si la somme de tous les compteurs vaut zéro, alors le nombre de messages envoyés est exactement égal au nombre de messages reçus — autrement dit, tous les messages en transit ont été délivrés. Couplée à la condition « tous les processus inactifs », cette observation suffit à garantir la terminaison globale.

Algorithme 7.7 (Safra avec compteurs)**.** Chaque processus P_i maintient un compteur entier c_i, initialisé à 0, et une couleur locale **blanc** (inactif) ou **rouge** (actif).

- **Envoi d'un message par P_i :** $c_i := c_i + 1$.
- **Réception d'un message par P_i :** $c_i := c_i - 1$.

Le jeton circule sur l'anneau et porte une **somme accumulée** S (initialement 0) et une couleur (blanc ou noir).

1. **Initialisation.** P_1 (inactif) crée un jeton avec $S = 0$, couleur **blanc**.
2. **Transmission par P_i.**
   - Si P_i est rouge : attendre de devenir inactif (blanc).
   - Mettre à jour : $S := S + c_i$.
   - Si P_i était rouge depuis le dernier passage : jeton := noir.
   - Transmettre le jeton à P_{i+1}.

3. **Réception par P_1.**
   - Si le jeton est blanc, P_1 est inactif, et $S + c_1 = 0$: **terminaison détectée**.
   - Sinon : remettre le compteur à zéro, lancer un nouveau tour avec $S = 0$, blanc.

Theoreme 7.8 (Correction de Safra) – *L'algorithme de Safra est correct. La condition $\sum_{i=1}^{n} c_i = 0$ est équivalente à « aucun message n'est en transit » si tous les processus ont contribué leur compteur au jeton pendant le même tour.*

Formellement : la terminaison est détectée si et seulement si le calcul est globalement terminé.

Preuve. Chaque envoi incrémente un c_i de 1, et chaque réception décrémente un c_j de 1. Si tout message envoyé a été reçu, alors pour chaque message m envoyé par P_i et reçu par P_j, la contribution $+1$ à c_i et -1 à c_j se compensent. La somme globale $\sum c_i$ est donc nulle si et seulement si le nombre de messages reçus est égal au nombre de messages envoyés, c'est-à-dire si aucun message n'est en transit.

La condition de couleur blanche du jeton garantit qu'aucun processus n'a été réactivé après avoir transmis le jeton, de sorte que les compteurs collectés reflètent bien l'état au moment de la collecte. □

Remarque 7.9 – *L'avantage majeur de l'algorithme de Safra sur le jeton de Dijkstra est qu'il fonctionne dans un modèle **asynchrone** : les processus n'ont pas besoin de se synchroniser sur un cycle d'horloge commun. Les messages peuvent être retardés arbitrairement. Cela rend l'algorithme applicable à une classe beaucoup plus large de systèmes distribués réels. En contrepartie, la preuve de correction est plus délicate car les compteurs sont collectés à des instants différents.* ◇

7.4 Schéma de crédit de Mattern

L'algorithme de Mattern adopte une approche radicalement différente, inspirée de la comptabilité financière : chaque processus et chaque message dispose d'une fraction de **crédit**, et la terminaison est détectée quand l'initiateur a récupéré la totalité du crédit distribué dans le système.

Ce schéma est particulièrement adapté aux **calculs diffusants** (*diffusing computations*), c'est-à-dire les calculs qui démarrent à partir d'un seul processus initiateur et se propagent en créant dynamiquement des « sous-tâches » qui peuvent elles-mêmes en créer d'autres. L'arbre de calcul n'est pas connu à l'avance, ce qui rend inapplicables les approches basées sur une topologie fixe.

Algorithme 7.10 (Schéma de crédit de Mattern)**.** L'initiateur P_1 démarre avec un crédit total de 1.

1. **Envoi d'un message par P_i :** P_i divise son crédit courant par 2 ; il conserve la moitié et attache l'autre moitié au message.
2. **Réception d'un message par P_j :** P_j ajoute le crédit reçu à son crédit local.
3. **Fin de calcul de P_i :** P_i retourne son crédit résiduel à l'initiateur P_1 (par un message de retour).
4. **Détection.** Quand P_1 a collecté un crédit total égal à 1, le calcul est globalement terminé.

Intuition 7.11 – *Le schéma de crédit est une astuce comptable élégante : chaque fraction de crédit représente une « dette » d'un processus ou d'un message envers l'initiateur. Aussi longtemps qu'un message est en transit ou qu'un processus est actif, une fraction du crédit est « immobilisée ». Quand le crédit total revient à* 1*, cela signifie que toutes les dettes ont été remboursées — tous les processus ont terminé et tous les messages ont été reçus. La division par deux assure que le crédit total dans le système est toujours conservé (il se redistribue sans être créé ni détruit) et peut être arbitrairement petit pour les calculs très ramifiés, ce qui peut poser des problèmes de précision en virgule flottante. En pratique, on représente le crédit en notation exacte (fractions ou entiers binaires).* ◇

Les trois algorithmes présentés dans ce chapitre résolvent le même problème fondamental avec des compromis différents. Le jeton de Dijkstra–Safra est simple et efficace pour les anneaux synchrones. L'algorithme de Safra généralise au cas asynchrone au prix d'une légère complexité supplémentaire. Le schéma de crédit de Mattern est le plus général : il s'applique à n'importe quel calcul diffusant, sans supposer une topologie fixe ni un modèle synchrone. Le choix entre ces approches dépend des hypothèses du modèle et de la structure du calcul distribué considéré.

Instantanés globaux

§8

Un système distribué en cours d'exécution est, par nature, un objet difficile à observer. Chaque processus possède un état local qui évolue continuellement, et les canaux de communication transportent des messages dont ni l'expéditeur ni le destinataire ne connaît exactement la position dans le réseau à un instant donné. Pourtant, de nombreuses tâches essentielles — la vérification d'invariants globaux, la détection de blocages (*deadlocks*), la reprise sur erreur (*checkpointing*), ou encore le débogage — nécessitent de prendre une « photographie » cohérente de l'état global du système à un instant donné.

Le problème de l'**instantané global** (*global snapshot*) consiste précisément à capturer cet état de manière cohérente, c'est-à-dire de façon à ce que la photo résultante corresponde à un état que le système aurait pu effectivement traverser dans une exécution séquentielle. La difficulté fondamentale est qu'il est impossible d'arrêter le système pour le photographier : les processus continuent d'envoyer et de recevoir des messages pendant la procédure de capture. Il faut donc concevoir un protocole distribué qui coordonne la prise de photo **sans perturber le calcul en cours**.

Ce chapitre développe les notions de coupe cohérente et d'état global cohérent, puis présente deux algorithmes classiques : Chandy–Lamport pour les canaux FIFO, et Lai–Yang pour les canaux non-FIFO.

8.1 État global et coupe cohérente

Pour formaliser ce que signifie un « état global cohérent », nous devons d'abord introduire le vocabulaire des historiques de processus et des coupes.

Definition 8.1 (Historique d'un processus) – *L'**historique** du processus P_i, noté h_i, est la séquence (totalement ordonnée) de tous les événements que P_i a exécutés :*

$$h_i = (e_i^1, e_i^2, e_i^3, \ldots)$$

où e_i^k désigne le k-ième événement de P_i. Un événement est soit une action locale, soit un envoi de message, soit une réception de message.

Definition 8.2 (Coupe) – *Une **coupe** C d'un système à n processus est un tuple $C = (c_1, c_2, \ldots, c_n)$ où $c_i \in \mathbb{N}$ représente le nombre d'événements de P_i inclus dans la coupe. La coupe C sélectionne le préfixe $(e_i^1, \ldots, e_i^{c_i})$ de chaque historique local h_i.*

*L'**état global** associé à la coupe C est la collection des états locaux $(s_1^{c_1}, s_2^{c_2}, \ldots, s_n^{c_n})$ où $s_i^{c_i}$ est l'état de P_i après son c_i-ième événement.*

Une coupe sépare le passé du futur pour chaque processus, mais elle ne garantit pas en elle-même la cohérence. Un problème se pose quand un message semble avoir été reçu avant d'avoir été envoyé : cela signifie que l'envoi se situe **après** la coupe sur le processus émetteur, mais la réception se situe **avant** la coupe sur le processus récepteur. Un tel état global ne peut jamais avoir existé dans une exécution réelle.

Definition 8.3 (Coupe cohérente) – *Une coupe $C = (c_1, ..., c_n)$ est dite **cohérente** si et seulement si : pour tout couple d'événements e et f,*

$$f \in C \quad \text{et} \quad e \to f \quad \Rightarrow \quad e \in C.$$

Autrement dit, si un événement f est inclus dans la coupe et qu'un événement e précède causalement f, alors e est également inclus dans la coupe.

Cette condition peut s'énoncer de manière équivalente en termes de messages : une coupe est cohérente si et seulement si **aucun message envoyé après la coupe n'est reçu avant la coupe**. Cette formulation est plus opératoire pour concevoir des algorithmes.

Theoreme 8.4 (Caractérisation des coupes cohérentes) – *Une coupe $C = (c_1, ..., c_n)$ est cohérente si et seulement si : pour tout canal c_{ij} de P_i vers P_j, le nombre de messages envoyés sur c_{ij} et inclus dans C (c'est-à-dire envoyés lors des c_i premiers événements de P_i) est supérieur ou égal au nombre de messages reçus sur c_{ij} et inclus dans C (c'est-à-dire reçus lors des c_j premiers événements de P_j).*

Preuve. ($\Rightarrow$) Supposons la coupe cohérente. Soit m un message reçu par P_j lors de son k-ième événement, avec $k \leq c_j$ (réception dans la coupe). Alors la réception est dans C. L'envoi e précède causalement la réception f (règle de communication de $\to$). Par cohérence, $e \in C$, donc l'envoi a lieu parmi les c_i premiers événements de P_i. Tout message reçu dans la coupe a donc été envoyé dans la coupe.

($\Leftarrow$) Réciproquement, si tout message reçu dans C a été envoyé dans C, alors il n'y a aucun message envoyé après la coupe et reçu avant, ce qui est précisément la définition d'une coupe cohérente. □

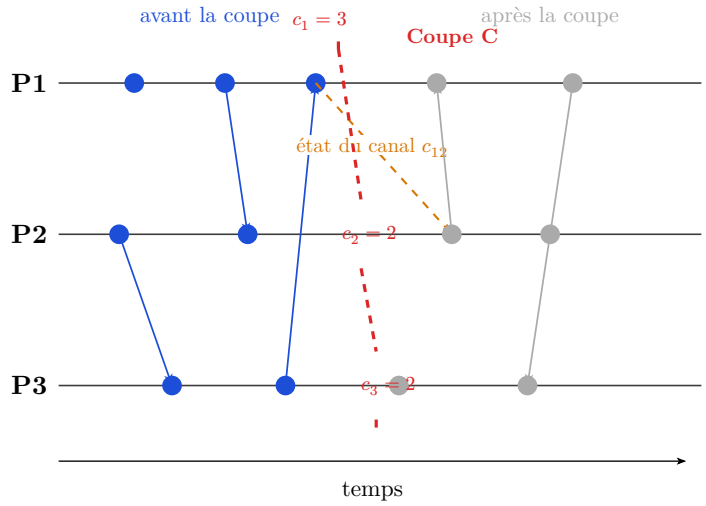
L'**état du canal** c_{ij} dans une coupe cohérente C est l'ensemble des messages envoyés sur c_{ij} dans la coupe mais pas encore reçus dans la coupe : ce sont les messages qui étaient en transit au moment de l'instantané.

Exemple 8.5 (Coupe cohérente et incohérente) – *Considérons trois processus P_1, P_2, P_3 avec les événements suivants : P_1 envoie m_1 à P_2 lors de son 3^e événement ; P_2 reçoit m_1 lors de son 2^e événement.*

- ***Coupe cohérente :*** *$C = (3, 2, 2)$ — l'envoi de m_1 (événement 3 de P_1) est inclus, et la réception de m_1 (événement 2 de P_2) l'est aussi. Cohérente.*
- ***Coupe cohérente avec message en transit :*** *$C = (3, 1, 2)$ — l'envoi de m_1 est inclus, mais la réception ne l'est pas ($c_2 = 1$). L'état du canal c_{12} contient m_1. C'est toujours cohérent : il n'y a pas de message reçu avant d'avoir été envoyé.*

- ***Coupe incohérente :*** $C = (2, 2, 2)$ — *la réception de* m_1 *(événement 2 de* P_2*) est incluse, mais l'envoi (événement 3 de* P_1*) ne l'est pas* ($c_1 = 2 < 3$)*. Cela correspond à un état global impossible.*

◇



Coupe cohérente : aucun message envoyé après la coupe n'est reçu avant. Le message en transit (en tirets ambrés) constitue l'**état du canal**.

Fig. 17. – Diagramme espace-temps illustrant une coupe cohérente. Les événements et messages **avant** la coupe sont en bleu ; ceux **après** sont en gris. Le message en tirets ambrés est envoyé avant la coupe (sur P_1) mais reçu après (sur P_2) : il constitue l'état du canal c_{12}. La coupe est cohérente car aucun message n'est reçu avant d'avoir été envoyé.

8.2 Algorithme de Chandy–Lamport (canaux FIFO)

L'algorithme de Chandy et Lamport (1985) est la première solution pratique au problème de l'instantané global. Il suppose que les canaux de communication sont **FIFO** : les messages émis sur un canal sont reçus dans l'ordre dans lequel ils ont été envoyés. Cette hypothèse est satisfaite par la grande majorité des protocoles de transport (TCP, par exemple).

L'idée directrice est d'utiliser des messages spéciaux appelés **marqueurs** pour délimiter la frontière de la coupe sur chaque canal. Un marqueur envoyé par P_i sur le canal c_{ij} signifie : « J'ai enregistré mon état ; tous les messages que j'ai envoyés sur ce canal **avant** ce marqueur font partie de l'instantané ; ceux envoyés **après** n'en font pas partie. » Grâce à la propriété FIFO, P_j sait exactement quels messages de c_{ij} précèdent la coupe (ceux arrivés avant le marqueur) et lesquels la suivent.

Algorithme 8.6 (Chandy–Lamport)**.** Chaque processus P_i maintient un ensemble $\mathrm{rec}(c_{ji})$ pour chaque canal d'entrée c_{ji} : les messages à inclure dans l'état du canal.

1. **Déclenchement (par n'importe quel processus P_i).**
   - P_i enregistre son état local s_i.
   - P_i envoie un **MARQUEUR** sur chacun de ses canaux de sortie.
   - P_i commence à enregistrer les messages reçus sur chacun de ses canaux d'entrée.
2. **Réception d'un MARQUEUR par P_i sur le canal c_{ji}.**
   - **Si P_i n'a pas encore enregistré son état :**
     - Enregistrer l'état local s_i maintenant.

- L'état du canal c_{ji} est **vide** (le marqueur arrive avant tout message post-coupe de P_j).
- Envoyer un MARQUEUR sur tous les canaux de sortie.
- Commencer l'enregistrement sur tous les autres canaux d'entrée.

- **Si P_i a déjà enregistré son état :**
  - L'état du canal c_{ji} est l'ensemble des messages reçus sur c_{ji} depuis l'enregistrement de l'état de P_i jusqu'à la réception de ce marqueur.
  - Arrêter l'enregistrement sur c_{ji}.

3. **Terminaison.** L'algorithme se termine quand tous les processus ont enregistré leur état et quand l'état de chaque canal a été déterminé.

Theoreme 8.7 (Correction de Chandy–Lamport) – *L'état global enregistré par l'algorithme de Chandy–Lamport est une **coupe cohérente** : il correspond à un état global que le système a pu effectivement traverser dans une exécution correcte.*

Preuve. Il suffit de montrer que l'état enregistré ne contient aucun message reçu avant son envoi. Soit m un message envoyé par P_i à P_j. Deux cas se présentent :

- **m est envoyé avant que P_i n'enregistre son état.** Alors m est envoyé avant le marqueur de P_i sur c_{ij}. Par FIFO, m arrive avant le marqueur chez P_j. Soit P_j enregistre son état à la réception du marqueur : m est arrivé avant, donc il est dans l'état du canal c_{ij} si P_j avait déjà enregistré son état, ou il est inclus implicitement dans l'état local de P_j sinon. Dans les deux cas, la réception de m est dans la coupe.
- **m est envoyé après que P_i a enregistré son état.** Alors m est envoyé après le marqueur de P_i. Par FIFO, m arrive après le marqueur chez P_j, donc après que P_j a enregistré son propre état. La réception de m n'est pas dans la coupe.

Dans les deux cas, il n'y a pas de message reçu dans la coupe mais envoyé hors de la coupe. La coupe est donc cohérente. □

Remarque 8.8 – *L'hypothèse FIFO est **indispensable** à la correction. Si les messages pouvaient se dépasser sur un canal, un message m envoyé après le marqueur pourrait arriver avant lui chez P_j. P_j enregistrerait alors m dans l'état du canal, alors que m est postérieur à la coupe sur P_i — produisant une coupe incohérente.* ◇

Exemple 8.9 (Trace de l'algorithme de Chandy–Lamport) – *Considérons trois processus P_1, P_2, P_3 dans un réseau quelconque.*

1. *P_1 décide d'initier le snapshot. Il enregistre son état local s_1, puis envoie un MARQUEUR sur c_{12} et sur c_{13}.*
2. *Avant l'arrivée du MARQUEUR, P_2 reçoit un message applicatif m_2 de P_1.*
3. *P_2 reçoit le MARQUEUR de P_1. Comme P_2 n'a pas encore enregistré son état, il enregistre s_2 maintenant. L'état du canal c_{12} est vide (le MARQUEUR est le premier message post-coupe). P_2 envoie un MARQUEUR sur c_{23}.*

4. *P_3 reçoit le MARQUEUR de P_1 en premier. Il enregistre s_3 et commence à enregistrer les messages reçus sur c_{23}.*
5. *P_3 reçoit le message m_3 de P_2 (envoyé avant le MARQUEUR de P_2). Ce message est ajouté à l'état du canal c_{23}.*
6. *P_3 reçoit le MARQUEUR de P_2. L'état du canal c_{23} est $\{m_3\}$.*

L'instantané global est (s_1, s_2, s_3) avec état de canal $c_{12} = \emptyset$, $c_{13} = \emptyset$, $c_{23} = \{m_3\}$. ◇

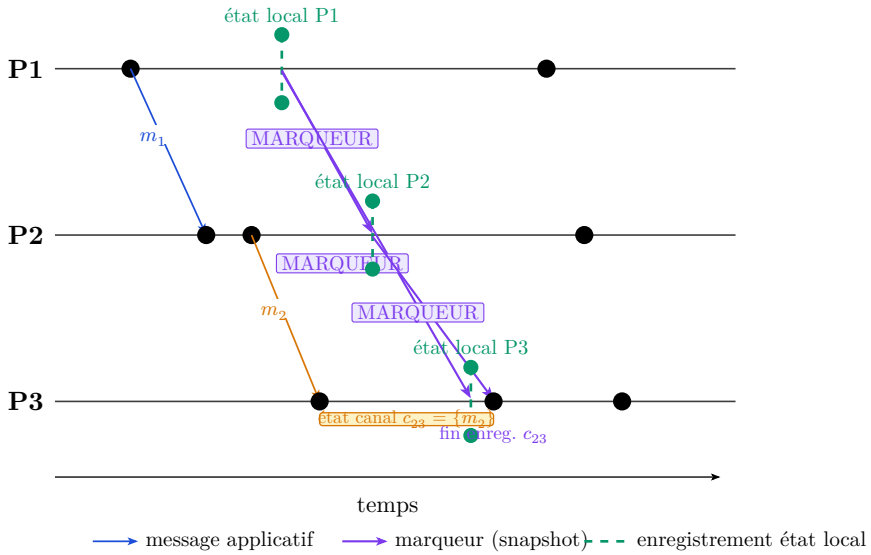


Fig. 18. – Trace de l'algorithme de Chandy–Lamport sur trois processus. Les lignes verticales pointillées vertes indiquent les moments d'enregistrement des états locaux. Les flèches violettes sont les MARQUEURS. Le rectangle ambré délimite l'état du canal c_{23} : le message m_2 envoyé par P_2 avant son enregistrement mais reçu par P_3 après que P_3 a enregistré son état.

8.3 Lai–Yang (canaux non-FIFO)

Lorsque les canaux de communication ne sont **pas FIFO**, l'approche de Chandy–Lamport échoue : un message m envoyé après le marqueur pourrait arriver avant lui chez le destinataire, brouillant la frontière de la coupe. L'algorithme de Lai et Yang (1987) contourne ce problème en supprimant entièrement les marqueurs explicites, au profit d'un mécanisme de **coloration** et de **piggybacking** (ajout d'informations aux messages applicatifs).

L'intuition est simple : avant le snapshot, les processus sont **blancs** ; après avoir décidé de participer au snapshot, ils deviennent **rouges**. Tout message envoyé par un processus **rouge** est lui-même marqué rouge. Lorsqu'un processus blanc reçoit un message rouge, il sait que l'expéditeur a déjà pris son snapshot : il doit donc prendre le sien avant de traiter ce message (pour que sa coupe soit cohérente avec celle de l'expéditeur).

Algorithme 8.10 (Lai–Yang)**.** Chaque processus est **blanc** au départ. Un processus qui a enregistré son état est dit **rouge**.

1. **Déclenchement du snapshot par P_i.**
   - P_i passe à l'état rouge et enregistre son état local s_i.
   - À partir de maintenant, tous les messages envoyés par P_i sont **rouges** (marqués en piggybacking).

2. **Envoi d'un message m par P_i.**
   - Si P_i est rouge : m est marqué rouge.
   - Si P_i est blanc : m est marqué blanc.
3. **Réception d'un message rouge par P_j (blanc).**
   - P_j enregistre son état local s_j **avant** de traiter m.
   - P_j passe rouge.
   - Le message m (rouge) n'est **pas** inclus dans l'état du canal (il est postérieur à la coupe de P_i).
4. **Réception d'un message blanc par P_j (rouge).**
   - Ce message a été envoyé avant que l'expéditeur devienne rouge, donc avant la coupe de l'expéditeur. Mais P_j est déjà rouge, donc ce message est en transit à travers la coupe.
   - P_j l'ajoute à l'état du canal c_{ij}.

Theoreme 8.11 (Correction de Lai–Yang) – *L'état global enregistré par l'algorithme de Lai–Yang est une coupe cohérente, même dans les systèmes à canaux non-FIFO.*

Preuve. Soit m un message envoyé par P_i à P_j. Quatre cas selon les couleurs à l'envoi et à la réception :

- **m blanc, P_j blanc lors de la réception.** m est envoyé et reçu avant les snapshots respectifs. L'envoi et la réception sont tous deux dans la coupe (ou tous deux hors de la coupe si les snapshots ont lieu après). Cohérent.
- **m blanc, P_j rouge lors de la réception.** m a été envoyé avant le snapshot de P_i, mais reçu après le snapshot de P_j. P_j ajoute m à l'état du canal : la coupe reconnaît que m était en transit. Cohérent.
- **m rouge, P_j blanc lors de la réception.** P_j reçoit un message rouge alors qu'il est encore blanc : il doit **d'abord** enregistrer son état avant de traiter m. Donc le snapshot de P_j est antérieur à la réception de m. m est donc reçu après la coupe de P_j. Or m rouge est envoyé après la coupe de P_i. Pas de problème.
- **m rouge, P_j rouge lors de la réception.** Les deux snapshots sont antérieurs à l'envoi et à la réception de m. m est hors coupe. Cohérent.

Dans tous les cas, aucun message n'est reçu dans la coupe sans avoir été envoyé dans la coupe. La coupe est cohérente. □

Remarque 8.12 – *L'algorithme de Lai–Yang évite d'envoyer des messages MARQUEUR supplémentaires sur chaque canal (ce qui nécessiterait $O(n^2)$ messages supplémentaires pour un graphe complet). En attachant les informations de couleur aux messages applicatifs (*piggybacking*), le surcoût en messages est nul : seul un bit de couleur est ajouté à chaque message. En revanche, l'algorithme requiert que chaque processus conserve en mémoire les messages blancs reçus après son passage au rouge, jusqu'à ce que tous les expéditeurs potentiels soient devenus rouges.* ◇

Intuition 8.13 – *La couleur dans Lai–Yang joue le même rôle conceptuel que le marqueur dans Chandy–Lamport : elle délimite la frontière de la coupe sur chaque canal. La différence est que dans Chandy–Lamport, le marqueur est un message* ***explicite*** *qui se déplace dans le canal, tirant parti de l'ordre FIFO ; dans Lai–Yang, la couleur est une information* ***implicite*** *attachée à chaque message, ce qui permet de fonctionner sans hypothèse d'ordre sur les canaux.* ◇

Les deux algorithmes présentés dans ce chapitre illustrent un principe général important en algorithmique distribuée : la même spécification (coupe cohérente) peut être réalisée par des mécanismes très différents, chacun adapté aux hypothèses du modèle sous-jacent. Dans les systèmes réels, le choix entre ces deux algorithmes — ou leurs nombreuses variantes — dépend des garanties offertes par la couche réseau et du budget en messages supplémentaires que l'application peut se permettre.

Consensus et tolérance byzantine

§9

Le **consensus** est le problème le plus fondamental de l'algorithmique distribuée tolérante aux fautes. Son énoncé est d'une simplicité trompeuse : plusieurs processus démarrent chacun avec une valeur initiale, et ils doivent tous se mettre d'accord sur une même valeur finale. Cette tâche banale dans un système centralisé devient extraordinairement difficile dès que certains composants peuvent tomber en panne — qu'il s'agisse d'arrêts simples (*crash failures*) ou de comportements malveillants (*Byzantine failures*).

Les résultats présentés dans ce chapitre forment le cœur théorique des systèmes distribués tolérants aux fautes. Nous verrons d'abord la définition formelle du consensus et ses exigences précises, puis le célèbre problème des généraux byzantins, avant d'établir l'impossibilité fondamentale de Fischer, Lynch et Paterson (FLP) dans les systèmes asynchrones. Nous terminerons par l'algorithme Flood-Set, qui montre que le consensus est soluble dans les systèmes synchrones.

9.1 Le problème du consensus

Formellement, le problème du consensus implique n processus $P_1, P_2, ..., P_n$, chacun possédant une valeur initiale $v_i \in \{0, 1\}$ (ou plus généralement dans un domaine V). Les processus communiquent par échange de messages, et l'objectif est que chaque processus finisse par **décider** (*decide*) une valeur, de sorte que toutes les décisions soient identiques et reflètent les entrées initiales.

Definition 9.1 (Problème du consensus) – *Un algorithme de consensus doit satisfaire les trois propriétés suivantes pour tous les processus **corrects** (non défaillants) :*

1. ***Accord (*Agreement*).** Deux processus corrects quelconques décident la même valeur : si P_i décide d_i et P_j décide d_j, alors $d_i = d_j$.*
2. ***Validité (*Validity*).** La valeur décidée est la valeur initiale de l'un des processus corrects. En particulier, si tous les processus corrects ont la même valeur initiale v, ils doivent décider v.*
3. ***Terminaison (*Termination*).** Tout processus correct décide en un temps fini.*

Ces trois propriétés semblent raisonnables, voire minimales. La propriété d'accord garantit l'**utilité** du consensus : tous les processus coordonnés aboutissent au même résultat. La propriété de validité empêche les solutions triviales — un algorithme qui décide toujours 0 satisferait l'accord, mais violerait la validité si les entrées sont toutes égales à 1. La terminaison assure que l'algorithme progresse effectivement.

Remarque 9.2 – *La difficulté fondamentale vient de la combinaison de ces trois propriétés face aux pannes. En leur absence, le consensus est trivial : chaque processus diffuse sa valeur, collecte toutes les valeurs, et prend le minimum. En présence de pannes, certains processus*

peuvent ne jamais répondre. Comment distinguer un processus lent d'un processus tombé ? Dans un système asynchrone, cette distinction est impossible — c'est précisément ce qu'exploite le résultat d'impossibilité FLP. ◇

9.2 Problème des généraux byzantins

Le modèle de panne le plus général — et le plus dangereux — est la **faute byzantine**. Un processus byzantin peut se comporter de manière arbitraire : il peut envoyer des messages contradictoires à des processus différents, envoyer des messages contenant des valeurs erronées, ou ne rien envoyer du tout. Ce modèle a été introduit par Lamport, Shostak et Pease (1982) sous la métaphore militaire des généraux byzantins : des généraux d'une armée doivent se coordonner pour attaquer ou se retirer, mais certains d'entre eux sont des traîtres qui cherchent à empêcher l'accord.

Definition 9.3 (Faute byzantine) – *Un processus est dit **byzantin** (ou **défaillant de manière arbitraire**) s'il peut s'écarter arbitrairement de son comportement spécifié. Un processus byzantin peut : envoyer des messages avec des contenus erronés, envoyer des messages différents à des destinataires différents pour le même événement, retarder ou omettre des messages, ou encore se coordonner avec d'autres processus byzantins pour compromettre le système.*

Le résultat central sur le consensus byzantin est une condition nécessaire et suffisante sur le nombre de processus.

Theoreme 9.4 (Condition nécessaire et suffisante — Lamport-Shostak-Pease (1982)) – *Dans un système de n processus dont au plus f sont byzantins, le consensus byzantin est **soluble** si et seulement si*

$$n \geq 3f + 1.$$

La preuve de ce théorème comporte deux parties : la borne inférieure (impossibilité pour $n \leq 3f$) et la borne supérieure (existence d'un protocole pour $n \geq 3f + 1$).

Borne inférieure : impossibilité pour $n = 3f$.

Supposons $n = 3f$ processus divisés en trois groupes A, B, C de taille f chacun. Supposons que le groupe C est byzantin. Voici le nœud de l'argument : du point de vue des processus de A, le groupe C byzantin peut se faire passer pour un groupe honnête ayant une valeur différente de celle que C présente à B. Plus précisément :

- C envoie « $v_C = 1$ » aux processus de A et B.
- C envoie « $v_C = 0$ » aux processus de D (dans une variante à 4 processus).

Les processus de A et B ne peuvent pas distinguer ce scénario d'un scénario où C est honnête et a réellement la valeur que C leur a communiquée. Avec seulement f processus dans chaque groupe, il est impossible de former une majorité fiable pour démasquer les traîtres. On peut construire formellement une contradiction : si un algorithme tolère f fautes byzantines avec $n = 3f$, on peut trouver deux exécutions indiscernables par certains processus mais où ces processus doivent décider des valeurs différentes pour satisfaire validité et accord — une contradiction.

Borne supérieure : protocole avec $f+1$ tours pour $n \geq 3f+1$.

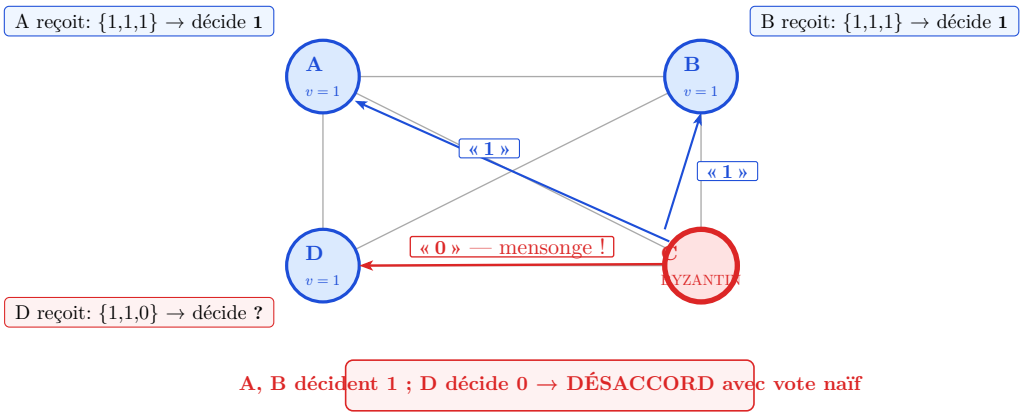
Pour $n \geq 3f+1$, il existe des protocoles qui atteignent le consensus byzantin. L'idée générale est de faire tourner $f+1$ rounds d'échange d'informations : dans chaque round, chaque processus diffuse toutes les valeurs qu'il a collectées jusqu'alors. Après $f+1$ rounds, les processus corrects ont suffisamment de redondance pour distinguer les informations cohérentes (venant de processus corrects) des informations incohérentes (venant de processus byzantins), et peuvent appliquer un vote majoritaire.

Exemple 9.5 (Cas n=4, f=1) – *Considérons $n = 4$ processus : A, B, D (honnêtes, avec $v = 1$) et C (byzantin). C envoie « 1 » à A et B, mais envoie « 0 » à D.*

Round 1 — vote naïf. *Chaque processus diffuse sa valeur initiale, puis effectue un vote majoritaire sur les valeurs reçues.*

- *A reçoit : {A:1, B:1, C:1, D:1} → vote majorité → décide **1**.*
- *B reçoit : {A:1, B:1, C:1, D:1} → vote majorité → décide **1**.*
- *D reçoit : {A:1, B:1, C:0, D:1} → vote majorité → décide **1**.*

Ici l'accord est atteint par chance : C n'a pas semé assez de discorde. Mais si C coordonne ses mensonges différemment (envoie « 1 » à A, « 0 » à B et D), alors B et D peuvent aboutir à des décisions différentes. Un seul tour est insuffisant en général pour $f = 1$. Il faut $f + 1 = 2$ tours pour garantir l'accord. ◇



Condition nécessaire et suffisante : $n \geq 3f+1$ (ici $n = 4, f = 1 : 4 \square 3\times1+1=4 \square$, mais protocole en 1 tour insuffisant)

Fig. 19. – Scénario byzantin avec $n = 4$, $f = 1$. Le processus C (encadré en rouge) envoie « 1 » à A et B mais « 0 » à D. Avec un simple vote naïf en un tour, A et B peuvent décider différemment de D, violant la propriété d'accord. La condition $n \geq 3f+1$ est nécessaire mais un protocole multi-tours est indispensable.

9.3 Impossibilité FLP

Le résultat de Fischer, Lynch et Paterson (1985) est l'un des théorèmes les plus importants — et les plus surprenants — de l'informatique distribuée. Il établit qu'il est **impossible** de résoudre le consensus dans un système asynchrone, même si les seules pannes possibles sont des **crashs** (arrêts définitifs) et qu'une seule panne peut survenir.

Ce résultat peut sembler paradoxal : un seul processus peut tomber, les autres fonctionnent parfaitement, et pourtant aucun algorithme déterministe ne peut garantir la terminaison. La clé est l'**asynchronisme** : dans un système asynchrone, il n'existe aucune borne sur les délais de

transmission des messages. Un processus qui ne répond pas est-il tombé en panne ou simplement lent ? Il est impossible de le savoir.

Theoreme 9.6 (Impossibilité FLP (Fischer-Lynch-Paterson, 1985)) – *Dans un système distribué **asynchrone**, il n'existe aucun algorithme **déterministe** qui résout le consensus même avec au plus **une** panne de type crash.*

La preuve formelle repose sur la notion de **configuration bivalente** : une configuration est **bivalente** si les deux valeurs de décision (0 et 1) sont encore possibles à partir de cet état, selon la suite des événements futurs. Une configuration est **univalente** si une seule valeur de décision est encore accessible.

Esquisse de preuve.

On montre deux lemmes :

1. **Existence d'une configuration initiale bivalente.** Si une configuration initiale C_0 était univalente pour toutes les valeurs des processus, il suffirait de modifier la valeur d'un processus (possiblement en panne) pour passer d'une configuration 0-valente à une configuration 1-valente. On montre qu'il doit exister une configuration initiale bivalente en considérant des chemins entre configurations 0-valentes et 1-valentes.
2. **Depuis toute configuration bivalente, il existe un événement qui maintient la bivalence.** Un algorithme qui tente de décider doit passer d'une configuration bivalente à une configuration univalente. Mais quel que soit le prochain événement exécuté (réception d'un message), on peut construire un ordonnancement des événements qui maintient la bivalence, en retardant indéfiniment le message déterminant. L'asynchronisme permet précisément ce report : il est toujours possible de prétendre qu'un message est simplement retardé plutôt que perdu.

La combinaison de ces deux lemmes montre qu'un algorithme déterministe ne peut jamais forcer la terminaison : il y aura toujours des exécutions où l'algorithme est maintenu indéfiniment dans une configuration bivalente.

Remarque 9.7 – *Le théorème FLP explique pourquoi tous les algorithmes de consensus pratiques — Paxos de Lamport, Raft de Ongaro et Ousterhout, Zab de Zookeeper — reposent sur des hypothèses supplémentaires qui sortent du cadre purement asynchrone :*

- ***Synchronie partielle*** *(*partial synchrony*) : il existe une borne sur les délais, mais elle n'est pas connue à l'avance.*
- ***Aléatoire** : certains algorithmes utilisent des bits aléatoires pour briser la symétrie et échapper aux configurations bivalentes.*

Dans ces modèles étendus, le consensus est soluble, mais FLP rappelle que cette solvabilité repose toujours sur des hypothèses non triviales concernant le modèle. ◇

Intuition 9.8 – *L'impossibilité FLP n'est pas un problème de complexité (« le problème est trop dur à calculer ») mais un problème de **calculabilité** dans un modèle spécifique. Il ne s'agit pas de trouver un algorithme plus efficace : **aucun** algorithme déterministe ne peut résoudre le consensus dans le modèle asynchrone avec crashs, quelle que soit sa complexité en temps ou en*

messages. C'est une limite fondamentale du modèle lui-même, comparable à l'indécidabilité du problème de l'arrêt pour les machines de Turing. ◇

9.4 Algorithme Flood-Set (synchrone, $f+1$ phases)

Le résultat FLP ferme la porte au consensus asynchrone, mais ouvre la voie à une question naturelle : le consensus est-il soluble dans un modèle **synchrone** ? La réponse est oui — et l'algorithme Flood-Set en donne une construction simple et élégante.

Dans le modèle synchrone, les processus s'exécutent en **rondes** : à chaque ronde, chaque processus envoie des messages, puis reçoit tous les messages envoyés pendant cette ronde (sauf ceux des processus crashés). Le système garantit une borne connue sur les délais. Les processus peuvent crasher (arrêt définitif), mais au plus f d'entre eux.

L'idée centrale du Flood-Set est d'inonder le réseau avec toutes les valeurs initiales connues. Après suffisamment de rondes, tous les processus vivants ont la même vue de l'ensemble des valeurs initiales et peuvent donc appliquer la même fonction déterministe pour décider.

Algorithme 9.9 (Flood-Set (Hadzilacos))**.** Chaque processus P_i maintient un ensemble W_i de valeurs, initialisé à $\{v_i\}$ (sa propre valeur initiale).

Pour les rondes $r = 1$ à $f + 1$:

1. **Diffusion.** P_i envoie W_i à tous les processus (y compris lui-même).
2. **Collecte.** P_i reçoit les ensembles W_j de tous les processus P_j qui lui ont envoyé un message pendant cette ronde.
3. **Mise à jour.** $W_i := \bigcup_{j \text{ reçu}} W_j$.

Décision. Après $f + 1$ rondes, P_i décide $\min(W_i)$ (ou toute fonction déterministe fixée à l'avance).

Theoreme 9.10 (Correction du Flood-Set) – *Après $f + 1$ rondes, tous les processus corrects ont le même ensemble W. En conséquence, ils décident tous la même valeur.*

Preuve. Nous montrons que si un processus correct P_i ajoute une valeur v à son ensemble lors de la ronde r, alors tout autre processus correct P_j aura v dans son ensemble à la fin de la ronde $r + 1$ (sauf si P_i crashe dans la ronde r).

Il y a au plus f crashs. Par le principe des tiroirs, sur $f + 1$ rondes, il existe au moins une ronde r^* sans aucun crash. Pendant la ronde r^*, tous les processus corrects reçoivent le message de tous les autres processus corrects. Donc à la fin de la ronde r^*, tous ont le même ensemble (l'union de tous les ensembles avant la ronde r^*).

Plus précisément : si une valeur v est dans W_i avant la ronde r^*, alors P_i la diffuse pendant r^* et tous les processus corrects la reçoivent. Si v est introduite pendant r^* par un processus P_k qui ne crashe pas (puisque r^* est sans crash), tous les processus la reçoivent. Donc après r^*, tous les processus corrects ont le même ensemble, et ils décident tous $\min(W)$. □

Proposition 9.11 (Complexité) – *L'algorithme Flood-Set effectue exactement $f+1$ rondes. À chaque ronde, chaque processus envoie son ensemble W_i à tous les n processus, soit $O(n)$ messages par processus et $O(n^2)$ messages par ronde. La taille de chaque message est $O(|W_i|)$ où $|W_i| \leq n$ (l'ensemble contient au plus n valeurs distinctes). La complexité totale en messages est donc $O(n^2 \cdot n) = O(n^3)$ valeurs transmises, sur $f+1$ rondes.*

Intuition 9.12 – *Le nombre $f+1$ de rondes est **optimal**. Avec f rondes seulement, il serait possible que f processus crashent, un par ronde, chacun au moment précis où il vient d'envoyer son ensemble à certains processus mais pas à d'autres. Cela peut créer une asymétrie d'information entre les processus survivants, empêchant l'accord. Avec $f+1$ rondes, on garantit qu'il y a au moins une ronde « propre » (sans crash) après laquelle toute l'information est uniformément distribuée.* ◇

En résumé, ce chapitre a établi les résultats fondamentaux du consensus distribué : la condition $n \geq 3f+1$ pour le consensus byzantin, l'impossibilité FLP dans le modèle asynchrone avec crashs, et la constructivité du Flood-Set dans le modèle synchrone. Ces résultats délimitent précisément ce qui est possible et ce qui ne l'est pas en matière de tolérance aux fautes, et constituent la base théorique indispensable à la conception des systèmes distribués robustes modernes.