Systèmes et algorithmes répartis

Convergence à terme

Philippe Queinnec

ENSEEIHT
Département Sciences du Numérique

25 octobre 2024

Plan

Contenu de cette partie

- CAP : conflit *Consistency – Availability – Partition*
- CRDT : *Convergent Replicated Data Type*
- CALM : *Consistency As Logical Monotonicity*
- Autostabilisation : convergence vers état légal

Plan

CAP – *Consistency, Availability, Partition tolerance*

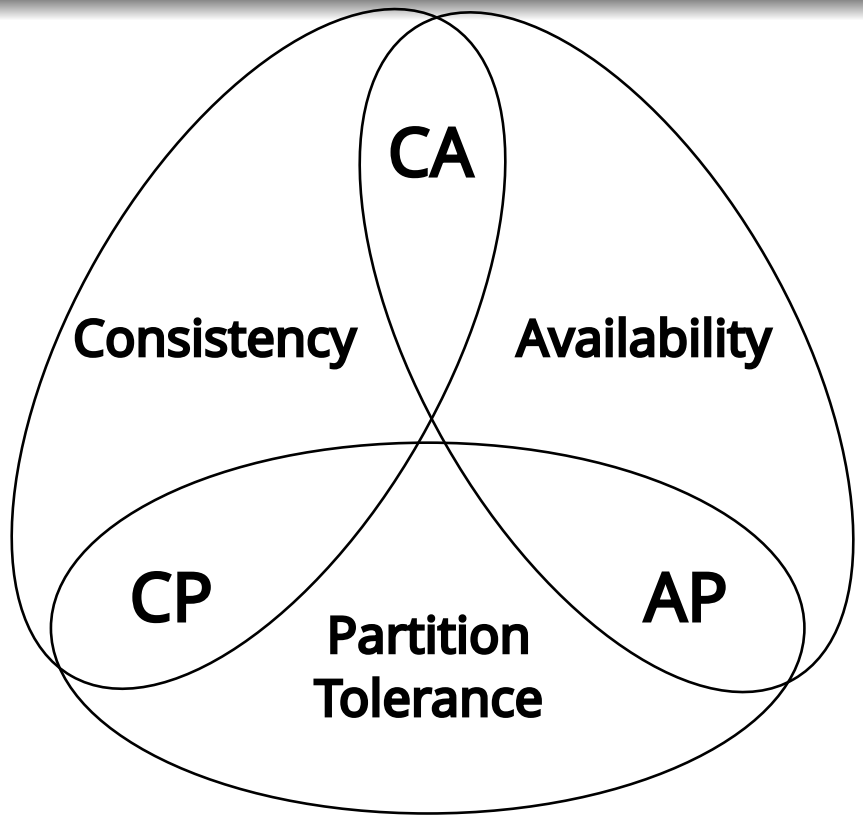
Considérons un système distribué avec données répliquées.

- C Cohérence forte (*Strong Consistency*) : tous les sites sont d'accord sur l'état du système = toute lecture renvoie la plus récente écriture (ou une erreur)
- A Disponibilité (*Availability*) : le système reste disponible (accessible aux clients) en présence de défaillance de sites
- P Tolérance aux partitions (*Partition Tolerance*) : le système continue à fonctionner correctement même si des sites se retrouvent isolés = si le réseau retarde ou perd un nombre arbitraire de messages

Théorème CAP : au plus deux parmi C, A, P sont possibles.

1. *Towards Robust Distributed Systems*, Eric Brewer. Symposium on Principles of Distributed Computing (PODC), 2000.
2. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, S. Gilbert, N. Lynch. ACM SIGACT News, 2002.

Théorème CAP



Exemples :

- CA : protocoles de permissions, de quorum strict
- CP : protocoles de quorum majoritaire
- AP : cohérence faible, résolution de conflit

CAP – preuve informelle

Considérons deux sites partageant un état.
Les deux sites se retrouvent dans des partitions différentes : ils ne peuvent pas communiquer.
Si une requête d'écriture arrive sur un site :

AP il la prend en compte, l'état n'est plus cohérent.

CP il la refuse pour maintenir l'état cohérent, le système n'est plus disponible.

Notes :

- Il n'est pas nécessaire de sacrifier C ou A s'il n'y a pas de partition.
- Un système CA considère l'impossibilité de partition (uniquement des défaillances de site par arrêt définitif)

Distributeur de billets – version CP

1. `Le distributeur contacte la banque`
2. `Le distributeur attend la réponse`
3. `Si la banque approuve le retrait, donner les billets sinon afficher le refus`

Défaillance du réseau $\rightarrow$ système indisponible

Distributeur de billets – version AP

```
1. Si la banque est accessible alors
2.     obtenir l'état du compte
3.     si la balance est insuffisante alors
4.         afficher le refus et abandonner
5.     finsi
6. finsi
7. donner les billets
8. enregistrer localement la transaction
9. Ultérieurement : envoyer la transaction à la banque
   pour réconciliation
```

→ cohérence à terme (*eventual consistency*)

Résolution de conflit

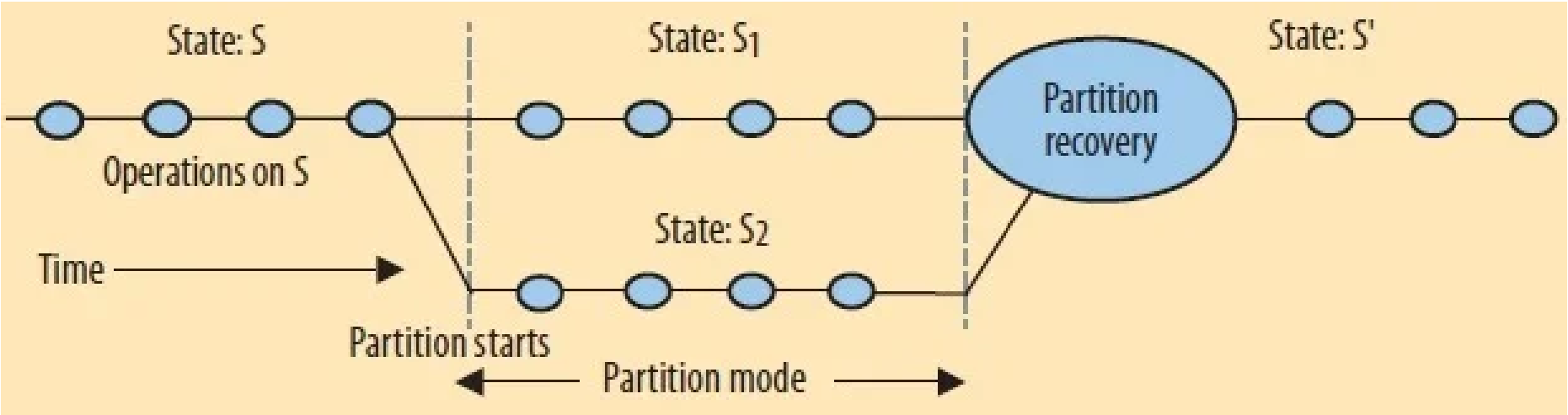
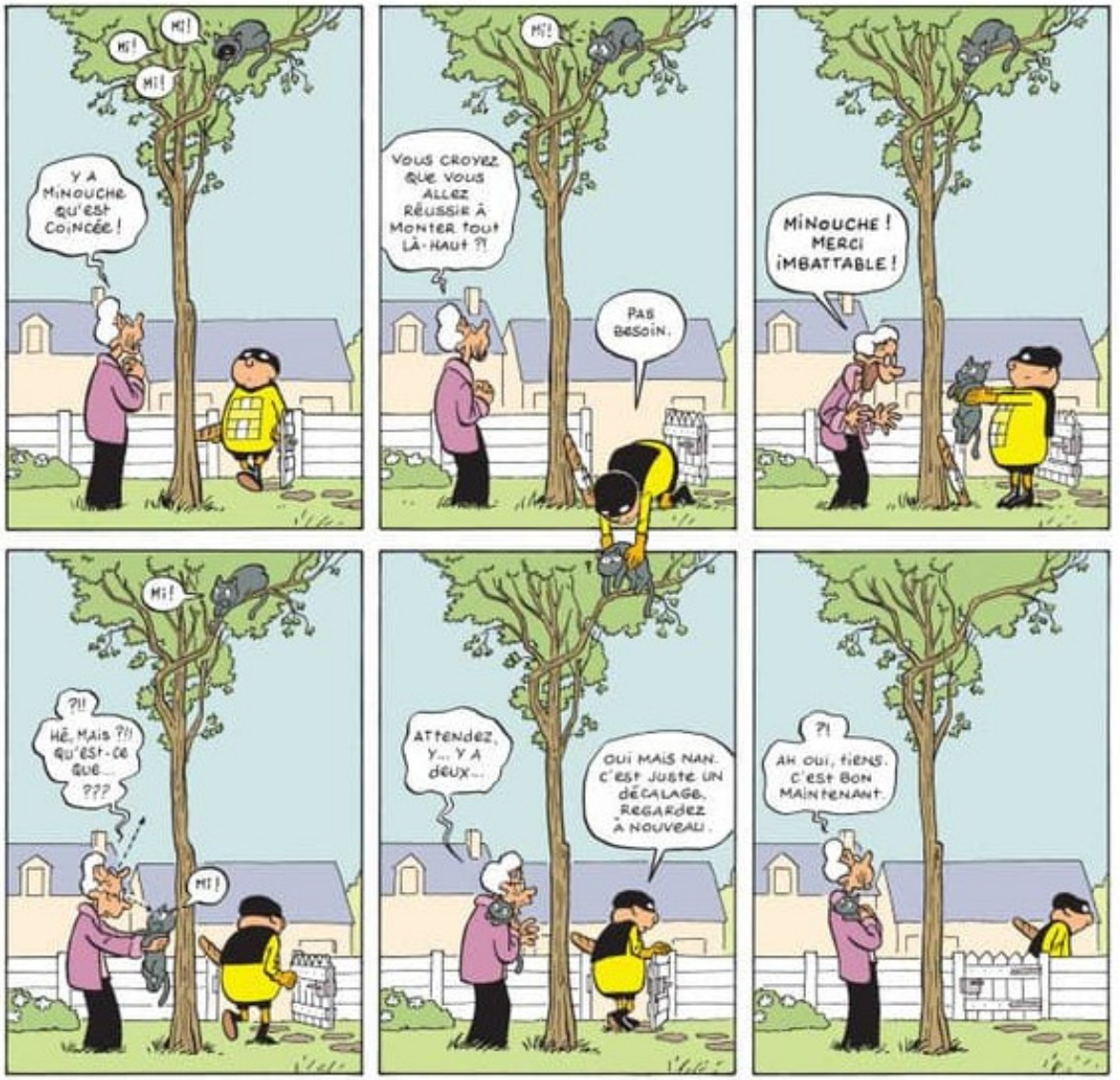


figure : Eric Brewer

Résolution de conflit



« Imbattable »
Pascal Jousselin
Dupuis 2017

Réconciliation

La réconciliation de partitions divergentes est spécifique à l'application :

- Billet d'avion : remboursement en €
- Paniers d'achat : faire l'union, avec le risque de voir revenir un item retiré, le client humain corrigera manuellement
- Distributeur de billets : rejeu ultérieur, avec limite sur la somme délivrée sans autorisation bancaire
- Logiciel de gestion de versions : vecteur de version (horloge vectorielle) ou écriture totalement ordonnées, plus réconciliation manuelle par le programmeur
- Types de données naturellement (re)convergents : CRDT

Plan

Observations sur la coordination

Inconvénients de la coordination (= la synchronisation) pour maintenir la cohérence de données répliquées :

- CAP : perte de disponibilité si cohérence stricte
- Peu performant, ne passe pas à l'échelle
- Complexe (cf consensus)

→ pourquoi ne pas faire de la réplication sans coordination ?

Type de données répliquées convergentes

CRDT : Commutative / Convergent Replicated Data Types

Donnée répliquée telle que :

- N'importe quel réplica peut être modifié indépendamment, concurremment et sans coordination avec les autres réplicas
- Un algorithme réconcilie automatiquement les réplicas
- Si les modifications cessent, les réplicas convergent à terme vers le même état

1. *Conflict-free Replicated Data Types*, Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. Stabilization, Safety, and Security of Distributed Systems, 2011

Exemple : compteur croissant (1/2)

- Objectif : un compteur global que l'on peut incrémenter (*inc*) et consulter (*get*)
- N sites qui peuvent $\pm$ communiquer entre eux
- Vecteur $1..N \to int$ répliqué sur chaque site
- *inc* : le site s incrémente $V_s[s]$
- *get* : $\sum_{i=1}^{N} V_s[i]$ (sous-estimation du compteur global)
- *Parfois*, un site envoie son vecteur à un autre site, qui le fusionne avec son vecteur en prenant le max deux à deux (anti-entropie).
- Si les incréments cessent, et que l'anti-entropie est vivace et équitable, les sites finissent par converger vers le même vecteur $\to$ la valeur du compteur global est connue de tous.

Les points importants sont que *inc* est commutatif et que l'état interne V_s est monotone : l'ordre des fusions est sans importance et la convergence à terme est garantie.

Exemple : compteur croissant (2/2)

Replica id
  state V : int[n] = [0,...,0] // *n = nombre de replicas*
  update inc()
    V[id] := V[id] + 1
  query get()
    return $\sum_i$ V[i]
  merge(X,Y)
    Z **suchthat** $\forall$ i : Z.V[i] = $\max$(X.V[i], Y.V[i])

Exemple : compteur généralisé (1/2)

- Opérations *inc* et *dec*, commutatives
- L'état global n'est plus monotone
- $\rightarrow$ utiliser deux compteurs globaux croissants, un pour les incréments (P), un pour les décréments (M)
- $\sum P[i] - \sum M[i]$ est une approximation du compteur global
- Convergence à terme de tous vers la même valeur exacte

Le compteur global n'est pas monotone mais l'état interne (P, M) est monotone.

Exemple : compteur généralisé (2/2)

Replica id
 state P, M : int[n] = [0,..,0] // *n = nombre de replicas*
 update inc()
  P[id] := P[id] + 1
 update dec()
  M[id] := M[id] + 1
 query get()
  return $\sum_i$ P[i] − $\sum_i$ M[i]
 merge(X,Y)
  Z **suchthat** $\forall$ i : Z.P[i] = max(X.P[i], Y.P[i])
  $\forall$ i : Z.M[i] = max(X.M[i], Y.M[i])

Opération de fusion

Cela fonctionne car la fusion m de deux états est

- Associative : $m(x_1, m(x_2, x_3)) = m(m(x_1, x_2), x_3)$
  Le groupement des opérations est sans importance.
- Commutative : $m(x_1, x_2) = m(x_2, x_1)$
  L'ordre d'application est sans importance.
- Idempotente : $m(x_1, x_1) = x_1$
  La duplication sans importance.

CvRDT basé état : Convergent Replicated Data Type

- Les sites s'échangent leur état local.
- Opérations associatives, commutatives, idempotentes.
- Les états forment un semi-treillis supérieur (*join semi-lattice*) : ordre partiel + existence d'une plus petite borne supérieure pour tout sous-ensemble.
- La fusion de deux états locaux (*merge*) est le *join* du semi-treillis, la modification (*update*) doit être monotone vis-à-vis de l'ordre partiel (toute modification accroît l'état).
- N'importe quelle stratégie de propagation des valeurs fonctionne, du moment qu'il n'y a pas de partition permanente (protocole *gossip*).

CmRDT basé opérations : Commutative Replicated Data Type

- Les sites propagent chaque opération de modification.
- La valeur locale est calculée par l'application des opérations reçues.
- Opérations commutatives, pas nécessairement idempotentes.
- La communication doit être de la diffusion fiable, sans duplication des messages. Certains CmRDT nécessitent que la diffusion soit causale.

CvRDT vs CmRDT

- CvRDT : plus simple pour raisonner (état disponible), faibles propriétés de la communication, inefficace si les objets sont gros
- CmRDT : plus complexe (prise en compte de l'histoire), messages plus petits mais diffusion fiable, éventuellement ordonnée
- Émulation d'un CvRDT par un CmRDT : l'opération transmise est le *merge* local, son application donne l'état
- Émulation d'un CmRDT par un CvRDT : l'état du CvRDT est l'histoire (causale) des opérations

Exemple : compteur généralisé en CmRDT

- Deux opérations *inc* et *dec* qui commutent
- Diffusion fiable des opérations → c'est bon !

```
Replica id
  state v
  query get()
      return v
  update inc()
      propagate    // diffusion asynchrone
      onreception
          v := v + 1
  update dec()
      propagate
      onreception
          v := v − 1
```

Registre

- Un registre est un objet avec deux opérations : *set* and *get*
- De nombreuses variantes existent selon la cohérence garantie en cas d'opérations concurrentes et distribuées
- Nombreuses utilisations : brique de base pour des structures plus complexes, ou donnée élémentaire : un fichier dans un système de fichier répliqué, un panier d'achat...

Registre *Last-Writer-Wins* (LWW), basé état (1/3)

- Un horodatage (*timestamp*) fournit une datation globale, totalement ordonnée, compatible avec la causalité : horloge de Lamport
- Chaque site conserve ⟨valeur du registre, date⟩, et transmet régulièrement à d'autres ce couple (à chaque changement, de temps à temps…)
- *set(v)* : couple ← ⟨v, nouvelle date⟩
- *get* : donne la valeur courante
- La fusion (réception d'un couple) conserve le couple avec la plus grande date
- Les dates ordonnent les couples → semi-treillis

Note : même principe en basé opération : la date permet d'appliquer ou d'ignorer un *set*.

Registre *Last-Writer-Wins* (LWW), basé état (2/3)

```
Replica id
  state x, t // x = valeur, t = timestamp
  update set(v)
      x, t := v, now()
  query get()
      return x
  merge(R1,R2)
      R suchthat if R1.t ≤ R2.t then R.x, R.t = R2.x, R2.t
                  else R.x, R.t = R1.x, R1.t
```

Registre *Last-Writer-Wins* (LWW), basé opération (3/3)

```
Replica id
  state x, t // x = valeur, t = timestamp
  query get()
      return x
  update set(v)
      t' := now()
      propagate(v,t')
      onreception(v,t')
          if t < t' then x, t := v, t'
```

Registre multi-valeurs, basé état (1/2)

- Utilisation d'horloges vectorielles
- Chaque site possède un *ensemble* de couples $\langle$valeur, date$\rangle$
- Cet ensemble est transmis régulièrement aux autres
- *set(v)* : ensemble $\leftarrow \{\langle$v, nouvelle date$\rangle\}$
- À la réception d'un couple $\langle$valeur, horloge$\rangle$, la fusion compare ce couple à chacun des couples de l'ensemble et :
  - conserve le couple ayant l'horloge max, si les horloges sont comparables
  - conserve les deux couples, si elles ne sont pas comparables
- *get* : peut fournir plusieurs valeurs (= plusieurs versions du registre)
- $\rightarrow$ convergence à terme vers le même ensemble partout. En outre, si tout ensemble non élémentaire finit par être réduit par l'applicatif (création d'une nouvelle valeur fusionnant les valeurs multiples), convergence vers une valeur unique.

Registre multi-valeurs, basé état (2/2)

Replica id $\in$ 0..N–1
  state S : set $= \{(\bot, [0,..,0])\}$ *// value, vector clock*
  let incVV() $=$
    let $\mathcal{V} = \{V \mid \exists x : (x, V) \in S\}$ **in**
    return [i $\in$ 0..N–1 $\mapsto$ **if** i $=$ id **then** $\max_{V \in \mathcal{V}}(V[i]) + 1$
      else $\max_{V \in \mathcal{V}}(V[i])$]
  update assign(v)
    S := {v, incVV()}
  query get
    return S
  merge(A,B)
    $A' \cup B'$
    where $A' = \{(x, V) \in A \mid \forall (y, W) \in B : V \parallel W \vee V \geq W\}$
      $B' = \{(y, W) \in B \mid \forall (x, V) \in A : W \parallel V \vee W \geq V\}$

Ensemble

Un ensemble possède une opération de consultation *lookup* et deux opérations de modification : *add*, *remove*. Elles ne commutent pas.

1. *Grow-Only Set* : pas de *remove*, proche du compteur croissant
2. *Two-Phase Set* : sous-ensembles d'ajout et de retrait avec contrainte de non-rajout d'un élément
3. *LWW-element-Set* : sous-ensembles d'ajout et de retrait avec datation des éléments
4. *Unique Set* : unicité des éléments + causalité de la communication
5. *Observed-Remove Set* : unicité des ajouts caché à l'utilisateur
6. ...

Ensemble : *Grow-Only Set*, *Two-Phase Set*

Grow-Only Set

Replica id
  state S : set of elements
  update add(e) : $S := S \cup \{e\}$
  query lookup(e) : **return** $e \in S$
  merge(X,Y) : Z **suchthat** $Z.S = X.S \cup Y.S$

Two-Phase Set

- Un sous-ensemble d'ajout (pour *add*) et un sous-ensemble de retrait (pour *remove*)
- $lookup(e) \triangleq e \in A \wedge e \notin R$
- Ne fonctionne que si un élément enlevé ne peut pas être remis plus tard :
  $add(e); remove(e); add(e) \neq add(e); add(e); remove(e)$
- Ressemble au compteur généralisé mais grosse différence : *inc*/*dec* commutent, *add*/*remove* ne commutent pas

Ensemble : *LWW-element-Set*, *Unique Set*

LWW-element-Set

- un sous-ensemble d'ajout et un sous-ensemble de retrait
- chacun contient des couples ⟨val,date⟩ comme le registre LWW
- $lookup(e) \triangleq \exists t : (e,t) \in A \wedge \nexists t' > t : (e,t') \in R$

Unique Set

- chaque élément ajouté est unique (utilisation d'UUID ou d'horloges de Lamport)
- un seul ensemble local, modifié par *add* et *remove*
- utilisation de la diffusion fiable causale
- la causalité garantit que *remove(e)* aura lieu partout après *add(e)*

Ensemble : *Observed-Remove Set*

Replica id
  state S : set of pairs $\langle$element, unique–tag$\rangle$
  query lookup(e)
    return $\exists u : \langle e, u \rangle \in S$
  update add(e)
    propagate($\langle$e, unique_uuid()$\rangle$)
    onreception($\langle e, u \rangle$)
      $S := S \cup \{\langle e, u \rangle\}$
  update remove(e)
    propagate($\{\langle e, u \rangle \mid \exists u : \langle e, u \rangle \in S\}$)
    onreception(R)
      $S := S \setminus R$

- Utilisation d'identifiant unique, caché à l'utilisateur
- Communication causale pour assurer qu'un *remove(e)* arrive après tous les *add(e)* qu'il a observés
- En cas de *add(e)* concurrent avec un *remove(e)*, l'ajout l'emporte (le retrait n'a pas pu voir l'ajout)

Exemple : Panier d'achat

- Opérations non commutatives *add(e)*, *remove(e)*, *checkout()*.
- Utiliser un *LWW-element-Set*, un *Unique Set* ou un *Observed-Remove Set* pour les opérations *add* et *remove*.
- *checkout()* ne commute pas avec les deux autres
  $\rightarrow$ requête *get()* causalement ordonnée et remplacer *checkout()* par *checkout*$(e_1, \ldots, e_n)$ qui fige le panier.
- C'est le client qui construit l'état final !
- Présence d'un être humain (l'acheteur) qui vérifie le contenu du panier au paiement et détecte le cas où *checkout(...)* n'a pas les bons objets et/ou procédure de remboursement / bon d'achat si l'acheteur détecte l'erreur ultérieurement.
  (le cas où *checkout* n'a pas les bons objets peut se produire en cas de défaillance, qu'un ajout / retrait n'a pas été propagé à tous les réplicas et que *get* interroge un réplica pas à jour $\rightarrow$ cas rare)

Autres structures

On connaît des CRDT pour

- les compteurs
- les ensembles
- les séquences et listes
- les arbres
- les *maps*
- l'édition collaborative de texte
- les paniers d'achat
- les préférences utilisateurs
- ...

Voir `https://crdt.tech/`

CRDT : conclusion

Un CRDT est une structure de données

+ Simple à implanter (pas de coordination)
– Complexe à concevoir
+ Robuste :
  - Perte de message ? Pas grave, ça reviendra plus tard
  - Messages dans le désordre ? Sans importance
  - Arrêt de sites ? Pas de coordination → pas un problème
+ Efficace (actions locales)
+ Réconciliation simple d'états distincts (fusion)
– État local plus gros
– Lecture arbitrairement vieille
– Retour en arrière possible quand la lecture change de réplica

Plan

Logique monotone

Une logique est monotone si une démonstration ne peut pas être invalidée par l'ajout de nouvelles hypothèses :
$F \models H \Rightarrow \forall G : F, G \models H$

Toutes les logiques classiques (logique des propositions, logique des prédicats, logiques temporelles LTL ou CTL) sont monotones.
Il existe des logiques non monotones, par exemple certaines logiques épistémiques ou pour du raisonnement par défaut :

- Les oiseaux volent, sauf rares exceptions
- Titi est un oiseau, donc il vole
- Titi est un oiseau, Titi est un manchot, donc il ne vole pas

L'ajout de nouveaux faits peut *réduire* l'ensemble des faits inférés.

Problème monotone et absence de coordination

Problème monotone

Un problème est une fonction de $\mathcal{P}(E) \to \mathcal{P}(F)$, où E et F sont deux ensembles quelconques.
Un problème est monotone si pour deux entrées S et T tel que $S \subseteq T$, alors $P(S) \subseteq P(T)$

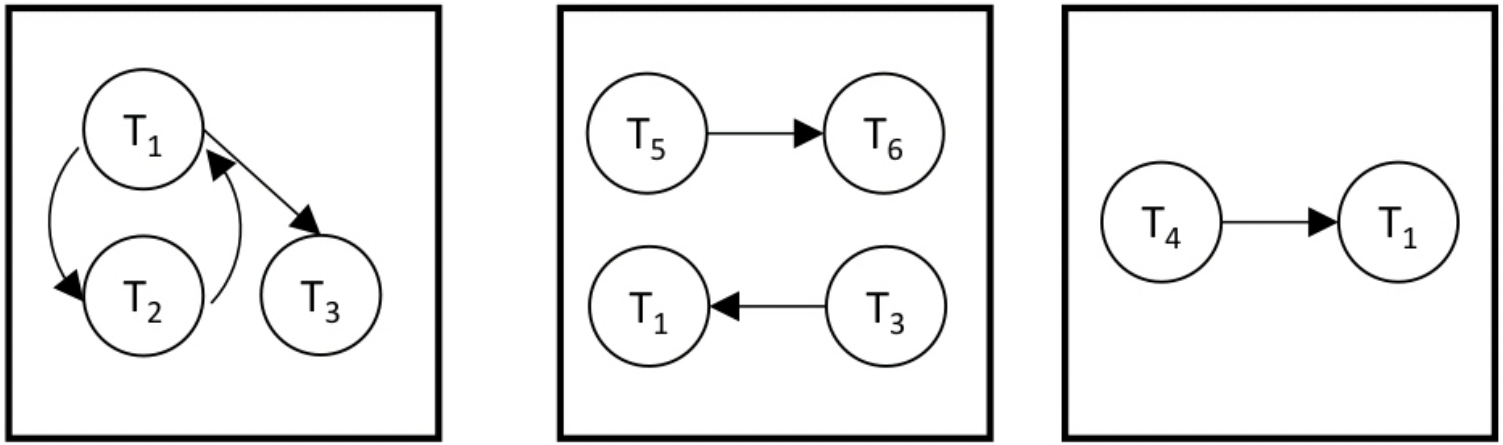
Théorème *Consistency as Logical Monotonicity* (CALM)

Un problème possède une implantation distribuée cohérente et sans coordination $\Leftrightarrow$ le problème est monotone

Noter la distinction problème / implantation : un problème monotone peut aussi posséder des implantations distribuées avec coordination

1. *Keeping CALM : when distributed consistency is easy*, Joseph M. Hellerstein and, Peter Alvaro, Communications of the ACM, 2020.

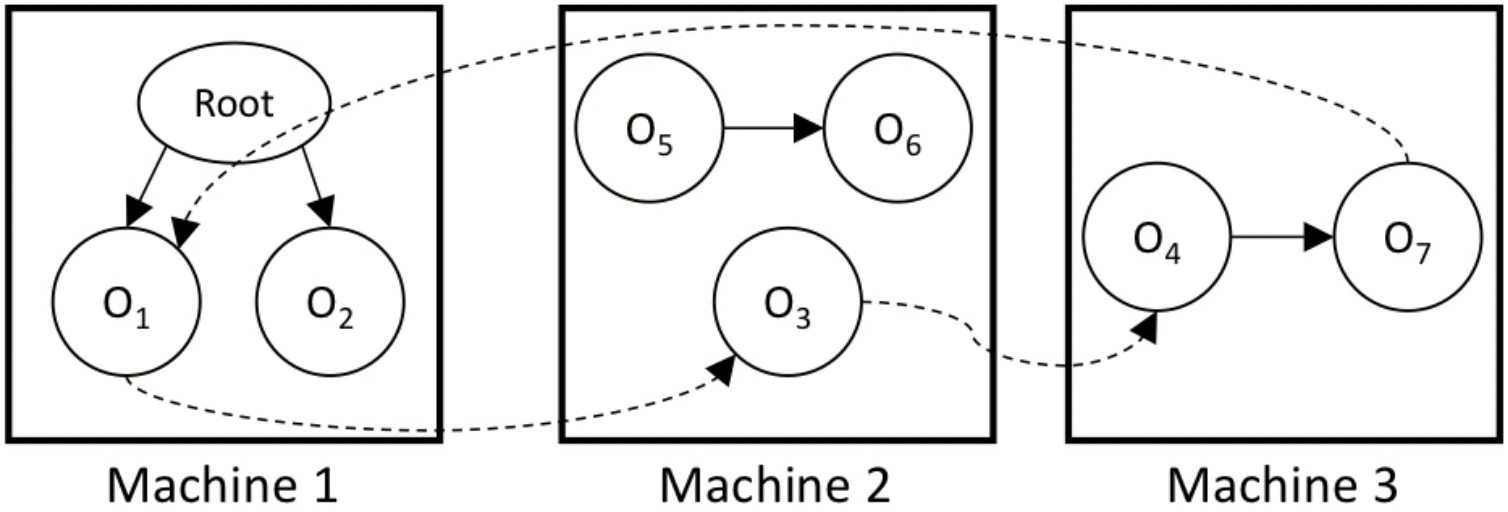
Un problème monotone : détection d'interblocage



Machine 1 Machine 2 Machine 3

- Des transactions réparties attendent l'accès à des ressources
- Les sites échangent les arcs qu'ils connaissent
- Quand un site constate un cycle, il décrète la présence d'un interblocage
- Le problème est monotone : si un cycle est détecté, la découverte de nouveaux arcs ne change pas la décision

Un problème non monotone : ramasse-miettes



- Des objets référencent d'autres objets. Un ensemble d'objets inaccessibles depuis une racine peut être supprimé
- Les sites échangent les arcs qu'ils connaissent
- Un site ne peut décider qu'un objet est inaccessible que s'il connaît tout le graphe (état global)
- Le problème n'est pas monotone : la découverte de nouveaux arcs change la décision d'inaccessibilité

Propriétés d'un problème monotone

- Les évolutions d'un problème monotone sont nécessairement commutatives : l'ordre des changements est sans importance sur le résultat final
- Il existe une caractérisation syntaxique d'un problème monotone s'il est exprimé en logique du premier ordre dans une algèbre relationnelle : opérations ensemblistes sauf $\setminus$, quantification $\exists$ mais pas $\forall$, mise à jour de relations par ajout mais pas de retrait...
  - Interblocage OK : $\exists(n, n) \in E^*$ (où E est l'ensemble des arcs connus par un site)
  - Ramasse-miettes KO : $\neg\exists(root, n) \in E^*$

1. *Keeping CALM : when distributed consistency is easy*, Joseph M. Hellerstein and, Peter Alvaro, Communications of the ACM, 2020.

Observations

- CAP exprime une impossibilité pour un système arbitraire. CALM restreint le type de système et obtient les trois propriétés C-A-P. : l'absence de besoin de coordination est équivalent à la disponibilité en présence de partitions.
- Un programme impératif (avec affectations) n'est pas monotone. Un programme fonctionnel est monotone. Le modèle MapReduce s'appuie sur des opérations monotones.

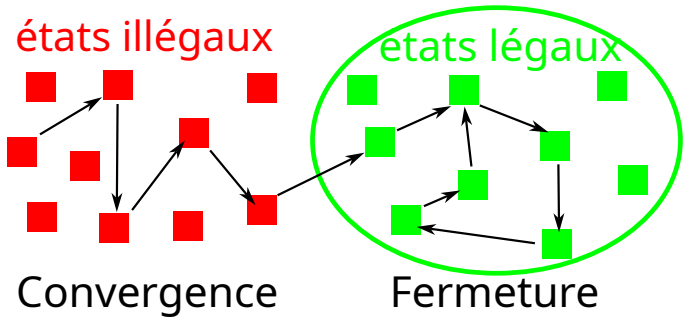
Plan

Autostabilisation

Système autostabilisant

Un système distribué est autostabilisant si, partant d'un état quelconque, il converge en temps fini vers un état légal, à partir duquel son comportement est correct.

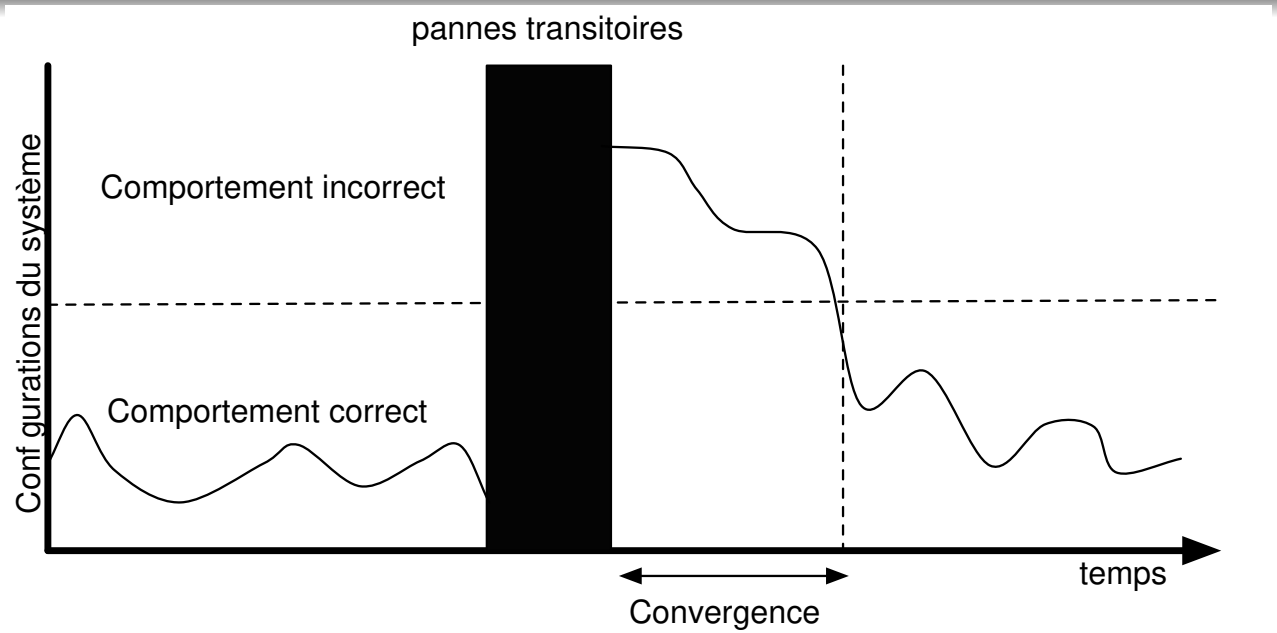
- Convergence : d'un état quelconque, le système finit par atteindre un état légal
- Fermeture : d'un état légal, le système reste dans un état légal



Un système autostabilisant ne termine jamais (un état terminal légal pourrait être l'état initial → zéro communication pour terminer → pas un algorithme distribué)

1. *Self stabilizing systems in spite of distributed control*, Edsger Dijkstra, Communications of the ACM, 1974.

Hypothèses



- État initial quelconque ≡ fautes transitoires
- État initial quelconque ≡ existence d'un adversaire
- L'adversaire ne peut pas modifier le *code* d'un site.
  L'adversaire peut modifier la valeur de n'importe quel message ou variable (y compris le compteur ordinal).
- Le graphe de communication reste connexe mais il peut y avoir des ajouts / départs de sites (selon l'algorithme)

Unicité sur un anneau (Dijkstra)

- processus $P_1 \dots P_N$ organisés en anneau
- une variable $x_p \in 0..N$ sur chaque processus
- un processus P_1 distingué
- état légal : un seul processus peut changer d'état

P_1 : **loop**
  if $x_1 = x_N$ **then**
    $x_1 := x_1 + 1 \bmod (N+1)$
P_i : **loop** $(i \in 2..N)$
  if $x_i \neq x_{i-1}$ **then**
    $x_i := x_{i-1}$

Finalement, P_1 atteint un état v qu'aucun processus n'avait initialement. À ce point, tous les autres finiront par avoir cette même valeur v. L'état est alors légal et les processus changeront de valeur dans l'ordre $P_1, P_2, P_3 \dots$

Ordonnanceur / démon

- Les algorithmes nécessitent l'existence d'un ordonnanceur qui détermine la ou les transitions à exécuter et qui peut être vu comme un adversaire cherchant à empêcher la stabilisation
- Cet ordonnanceur peut être :
  - synchrone : tous les sites font une transition simultanément
  - central : l'ordonnanceur décide d'un unique site qui fait une transition
  - distribué : chaque site fait des transitions asynchrones
- L'ordonnanceur peut être contraint :
  - borné : écart maximal du nombre de transitions entre deux sites
  - équitable : tout site finit par faire une transition
  - arbitraire : aucune garantie (s'il n'y a qu'un site qui peut faire une transition, il sera choisi)

Un ordonnanceur distribué équitable (tout site fait des transitions infiniment souvent) est raisonnable et souvent suffisant. Les preuves utilisent souvent un ordonnanceur synchrone, plus simple.

Arbre de recouvrement : principe

- Donnée : graphe connexe, avec connaissance de ses voisins
- Objectif : construire un arbre de recouvrement en largeur, ancré sur une racine donnée
- Chaque site maintient (variables) :
  - parent : l'identité du site parent
  - distance : sa distance à la racine (nombre de hops)
- Les sites échangent périodiquement leur distance à la racine (la racine envoie 0)
- Chaque site choisit comme parent un voisin avec la distance minimale et met à jour sa distance (choix déterministe si plusieurs minimaux, si on veut que l'arbre soit stable une fois la convergence réalisée)
- Après k itérations de tous, les sites à distance $\leq k$ de la racine sont stables

1. *Self-stabilization*, Shlomi Dolev. The MIT Press, 2000.

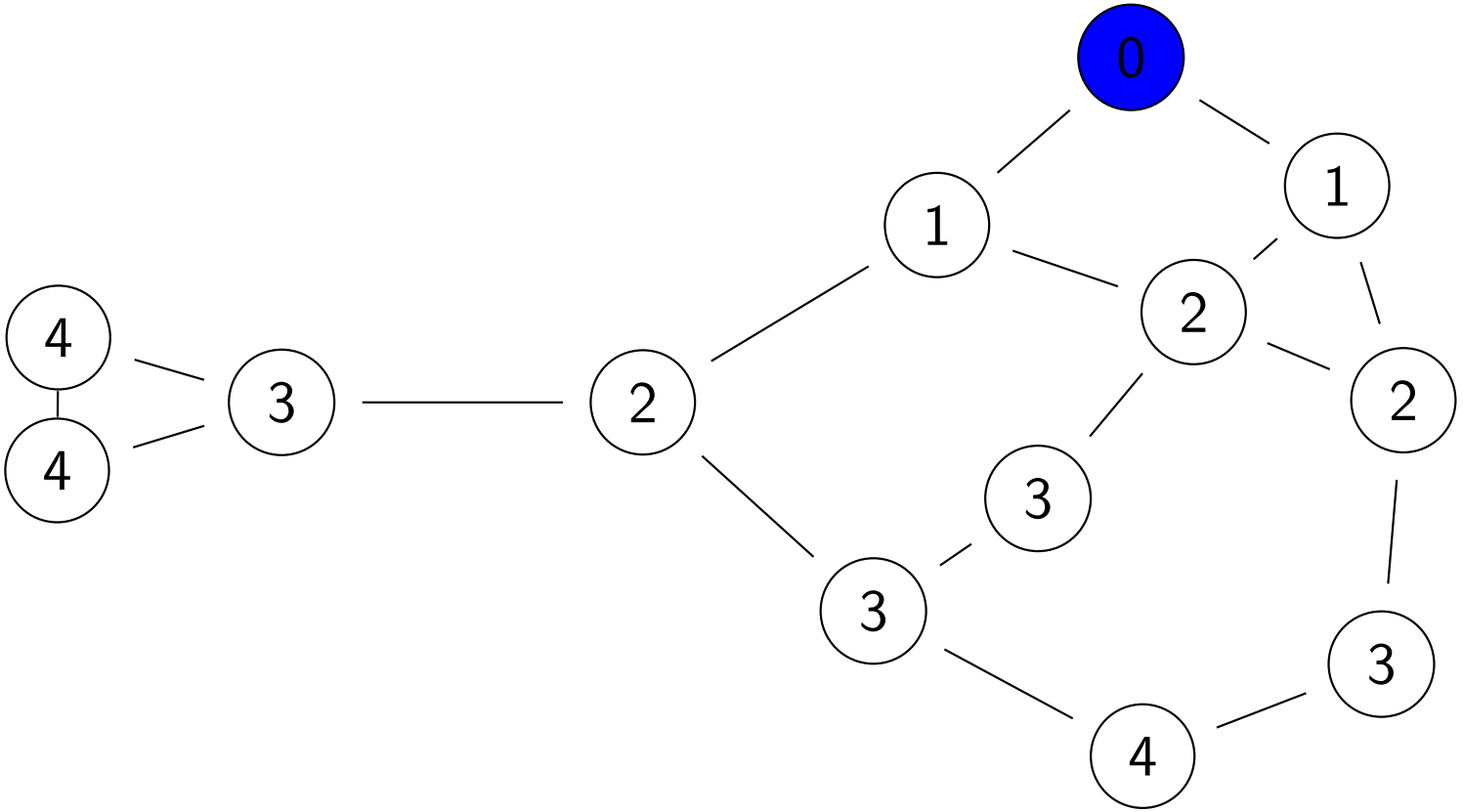
Arbre de recouvrement : code

```
Racine : loop
            parent := self
            envoyer à chaque voisin 0

Autre site : loop
                envoyer distance à chaque voisin
                attendre d'avoir reçu la distance de chaque voisin
                parent := choisir site avec distance minimale
                distance := 1 + cette distance minimale
```

- Nécessite un ordonnanceur équitable (tout site finit par faire un pas)
- Correct en cas d'ajout / départ de site, ou d'ajout / rupture de lien (tant que le graphe reste connexe)

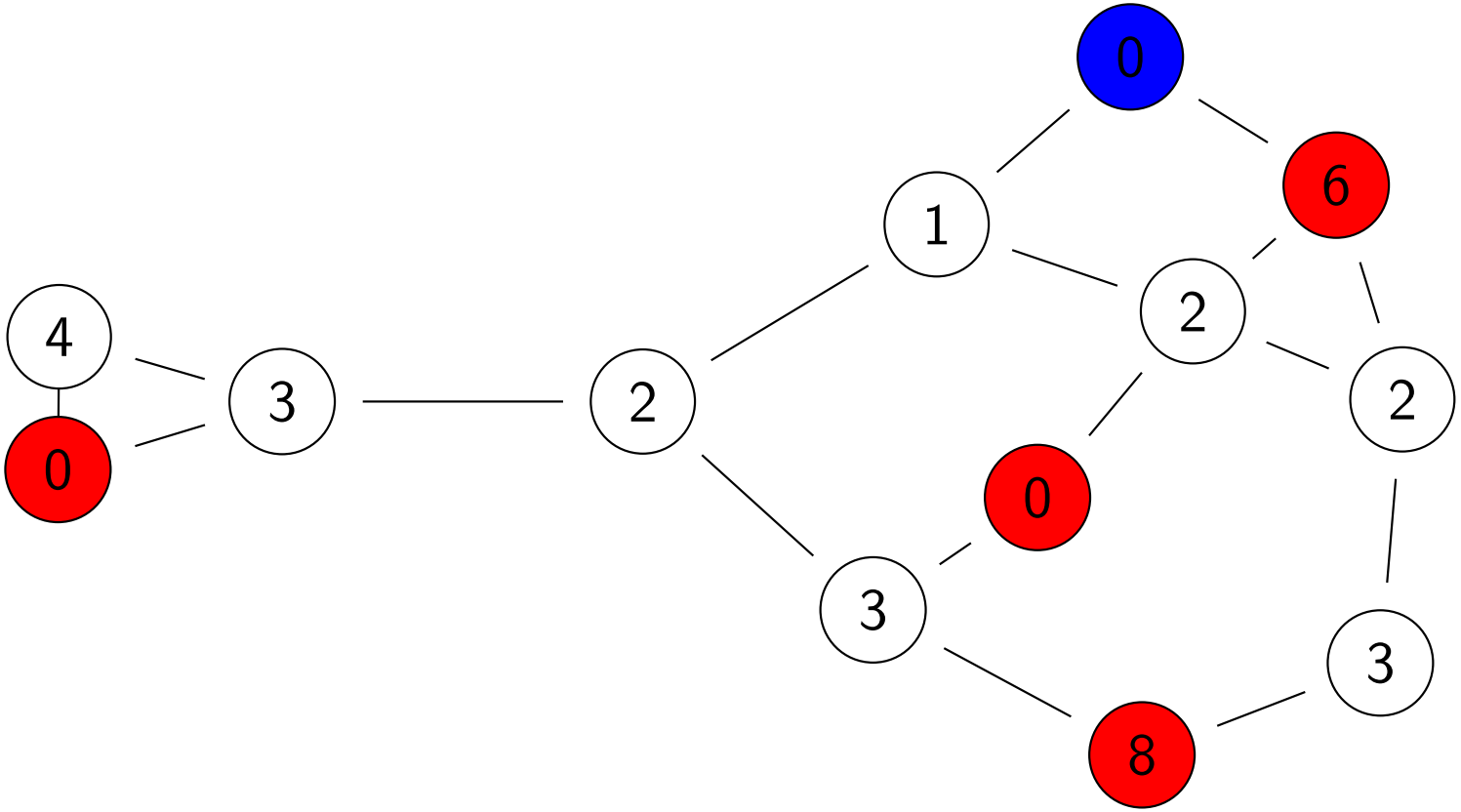
Arbre de recouvrement – exemple



racine, nœud stable, nœud erroné, nœud correct par hasard

Simulation avec un ordonnanceur synchrone (pour simplifier)

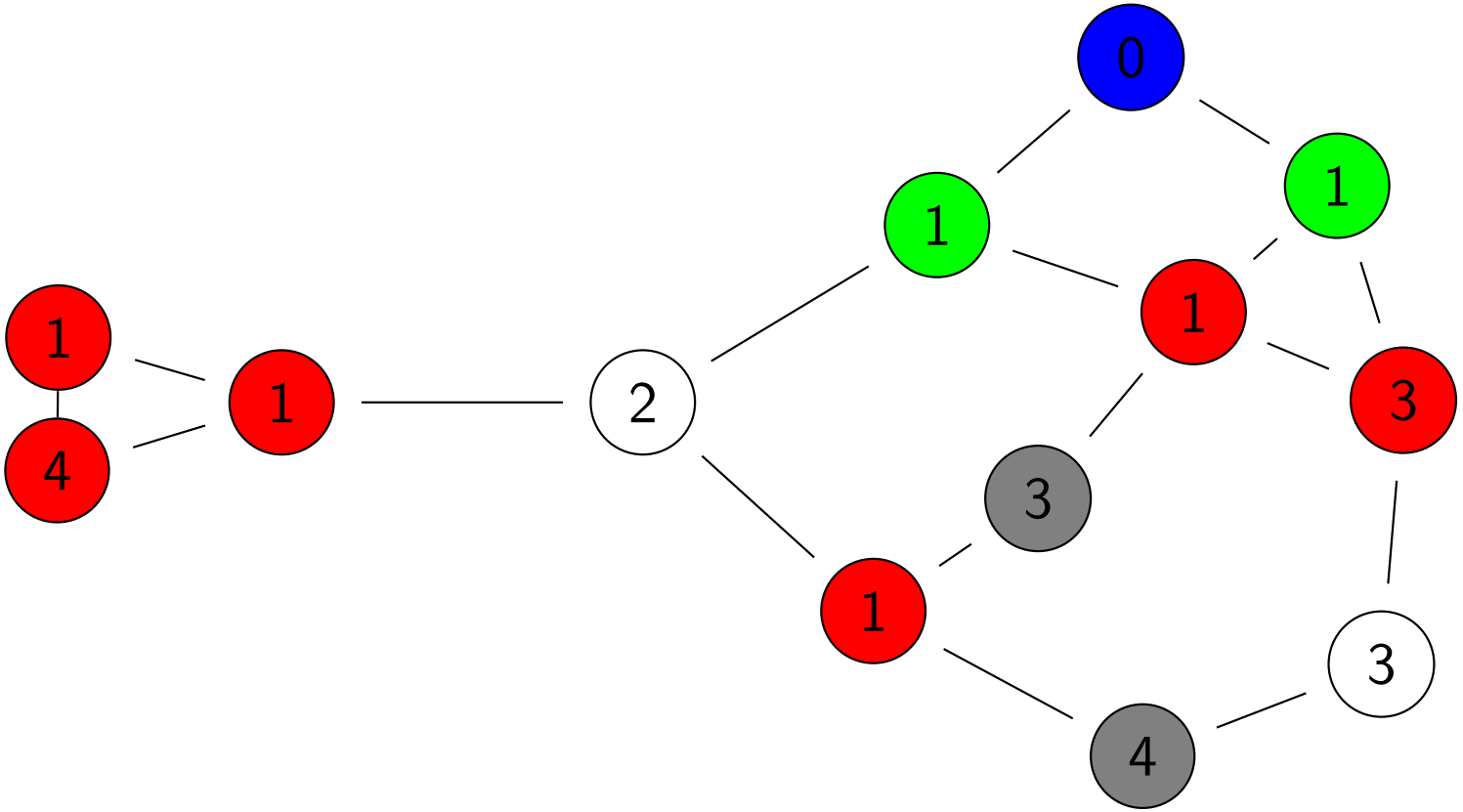
Arbre de recouvrement – exemple



racine, nœud stable, nœud erroné, nœud correct par hasard

Simulation avec un ordonnanceur synchrone (pour simplifier)

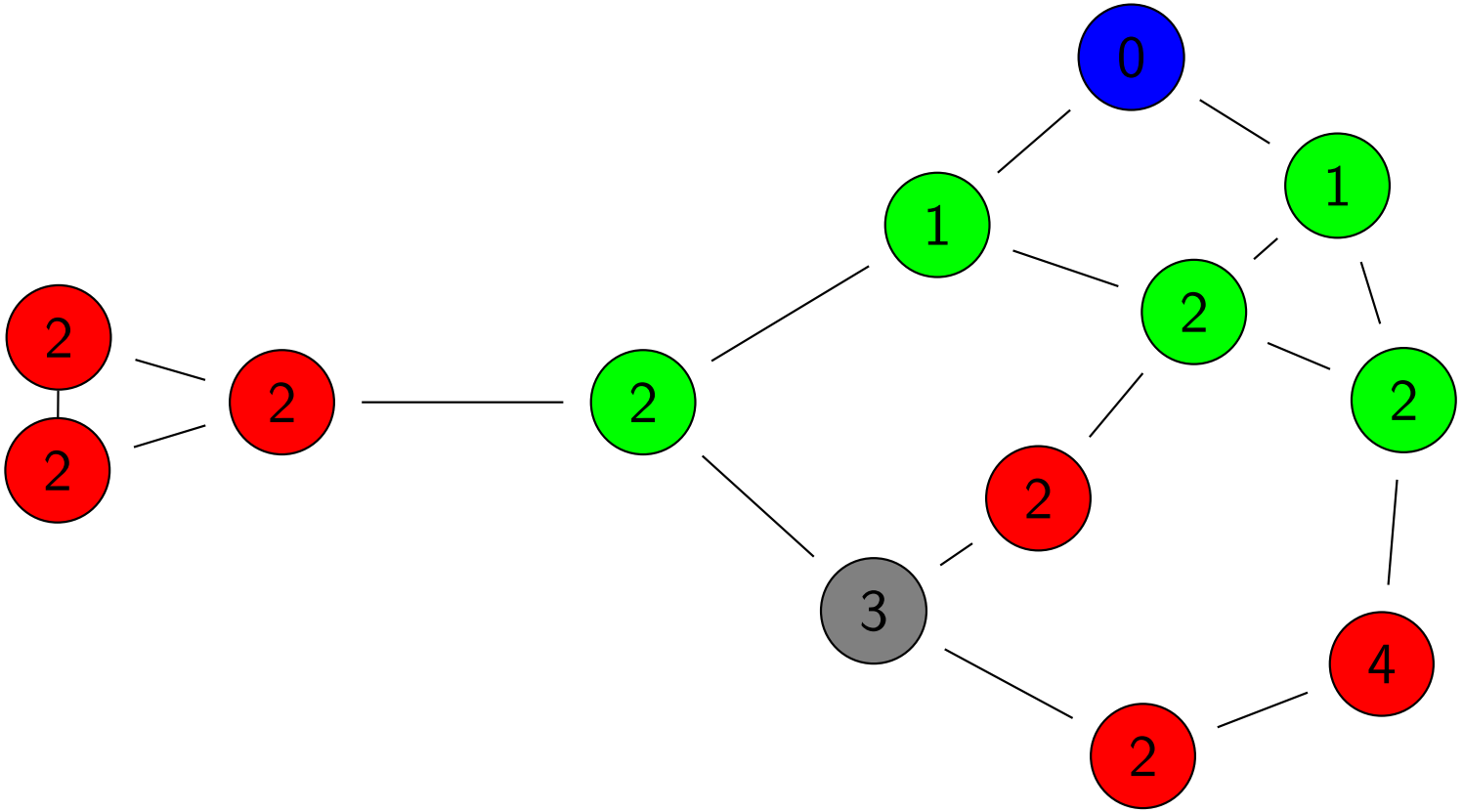
Arbre de recouvrement – exemple



racine, nœud stable, nœud erroné, nœud correct par hasard

Simulation avec un ordonnanceur synchrone (pour simplifier)

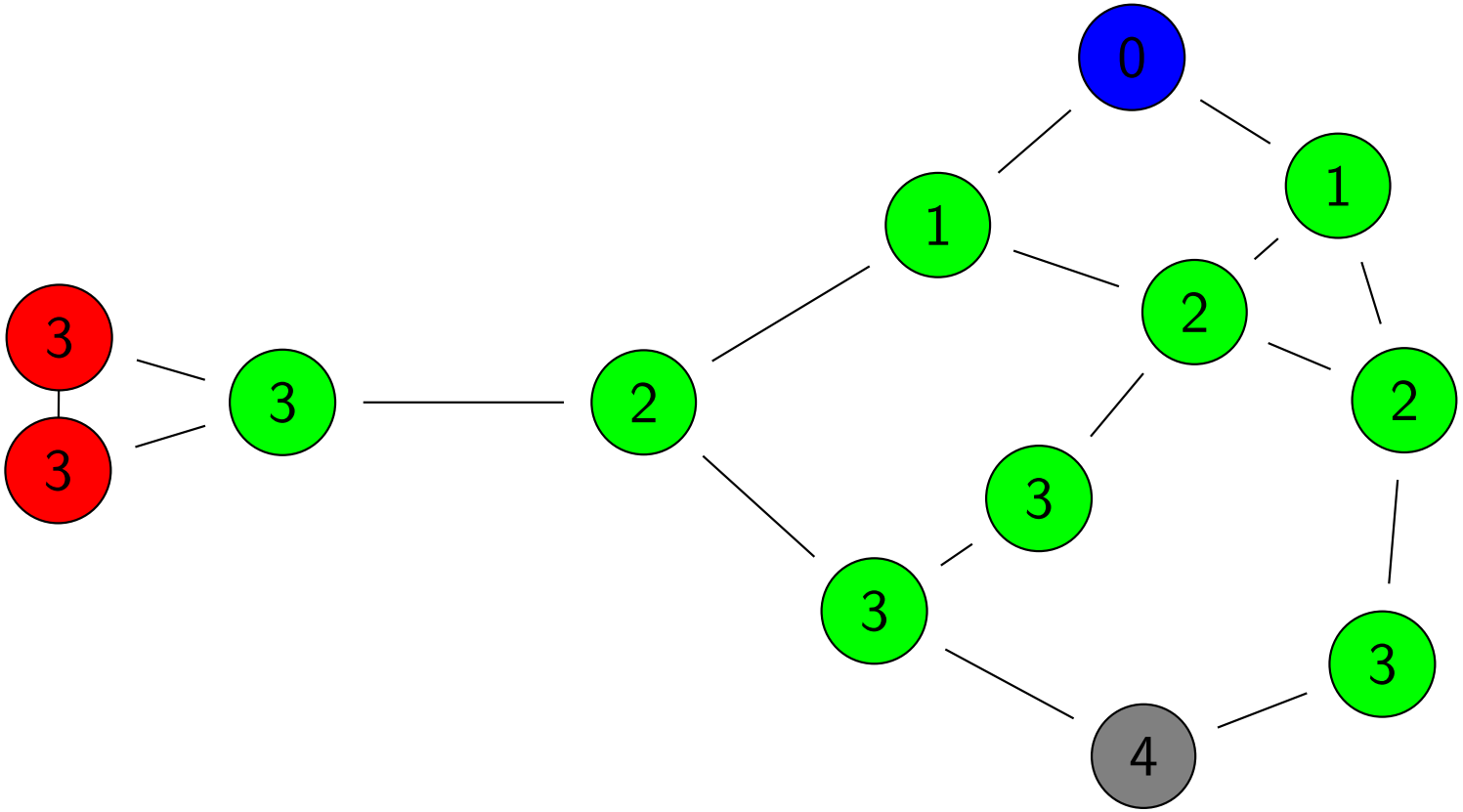
Arbre de recouvrement – exemple



racine, nœud stable, nœud erroné, nœud correct par hasard

Simulation avec un ordonnanceur synchrone (pour simplifier)

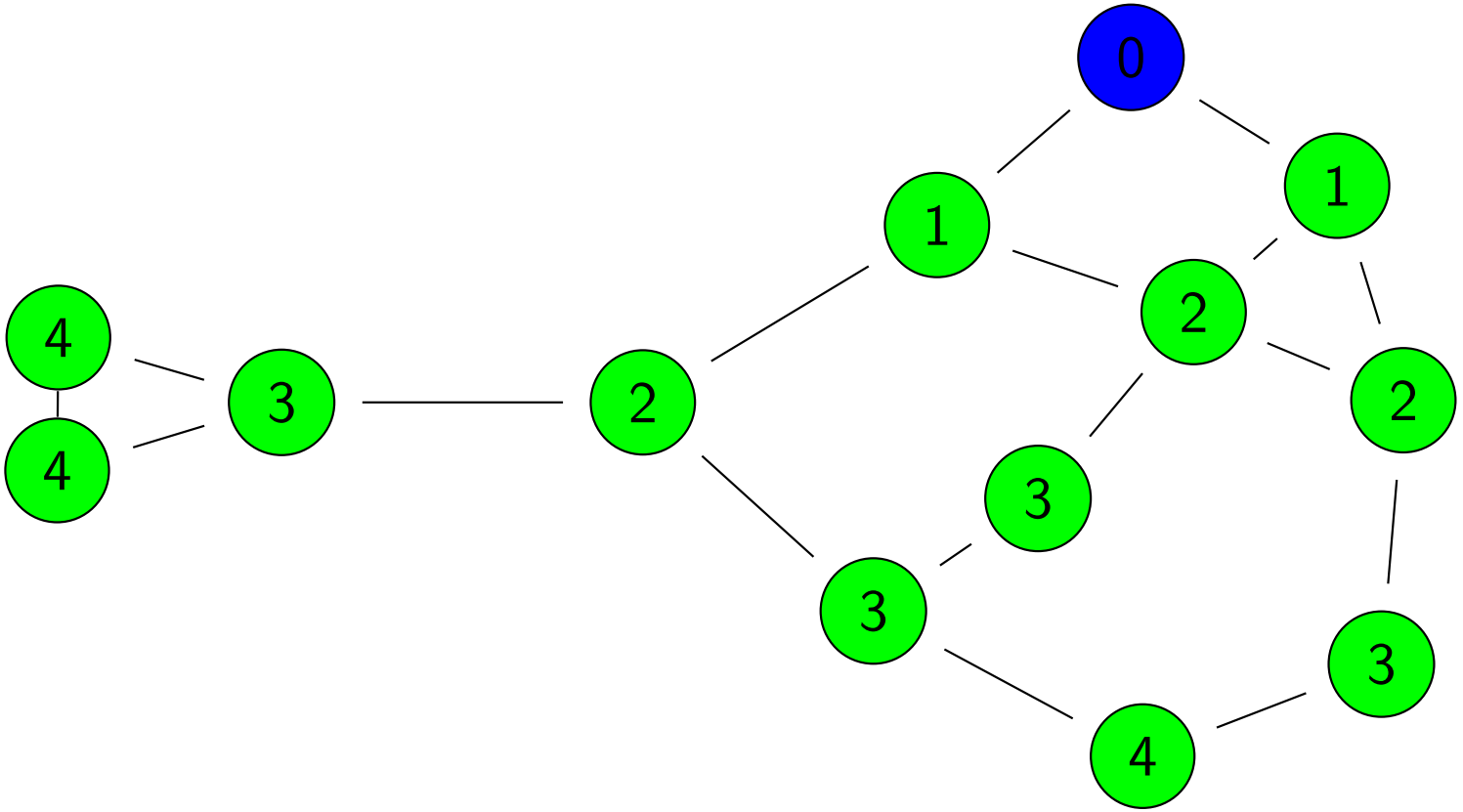
Arbre de recouvrement – exemple



racine, nœud stable, nœud erroné, nœud correct par hasard

Simulation avec un ordonnanceur synchrone (pour simplifier)

Arbre de recouvrement – exemple



racine, nœud stable, nœud erroné, nœud correct par hasard

Simulation avec un ordonnanceur synchrone (pour simplifier)

Mesures de complexité

Temps de convergence

On mesure le nombre de transitions du processus le plus lent, ou le nombre total de transitions effectuées

- Temps de convergence : nombre maximal de transitions pour atteindre un état légal depuis un état initial arbitraire
- Temps de réponse : nombre maximal de transitions pour atteindre un état particulier depuis un état initial spécifique

Nombre de messages

On mesure le nombre de messages pour converger dans le pire cas

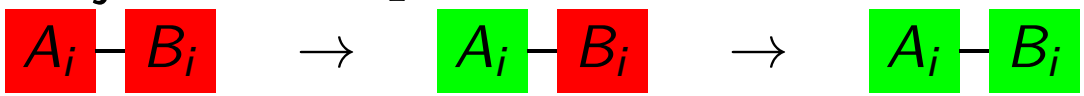
Avec un ordonnanceur équitable, l'algorithme d'exclusion mutuelle de Dijsktra converge en $O(n^2)$ tours. L'algorithme d'arbre de recouvrement converge en $O(diamètre)$ tours synchrones et $O(diamètre \times N \times V)$ messages (où V est la taille d'un voisinage).

Composition

La composition de deux algorithmes $\mathcal{AL}_1$ et $\mathcal{AL}_2$ est définie par :

- un algorithme $\mathcal{AL}$ réalise une tâche $\mathcal{T}$ si l'ensemble des traces légales est $\mathcal{T}$ (trace = séquence d'états)
- $\mathcal{AL}_1$ lit et construit des états A_i, $\mathcal{AL}_2$ lit des états $A_i \times B_i$ et construit des états $A_i \times B'_i$ en laissant la partie A_i inchangée
- composition équitable : alternance d'un pas $\mathcal{AL}_1$ et d'un pas de $\mathcal{AL}_2$ (en fait il suffit infiniment souvent chacun)
- si $\mathcal{AL}_1$ est autostabilisant pour $\mathcal{T}_1$ et si $\mathcal{AL}_2$ est autostabilisant pour $\mathcal{T}_2$ en utilisant $\mathcal{T}_1$, alors la composition équitable de $\mathcal{AL}_1$ et $\mathcal{AL}_2$ est autostabilisante

Note : $\mathcal{AL}_2$ ne peut pas savoir si $\mathcal{AL}_1$ est dans un état légal : $\mathcal{AL}_2$ se stabilisera le jour où $\mathcal{AL}_1$ sera stabilisé



Composition – exemple

Construction d'un algorithme autostabilisant d'exclusion mutuelle pour les graphes quelconques :

1. algorithme autostabilisant de construction d'un arbre de recouvrement
2. un parcours en profondeur de l'arbre forme un anneau (en ne gardant que la première occurrence des sites non feuilles)
3. algorithme autostabilisant d'exclusion mutuelle sur un anneau

Consensus et autostabilisation

On sait résoudre, de manière autostabilisante :

- les détecteurs de défaillance
- le consensus
- la réplication de machine à états
- (et au passage, la réinitialisation sans blocage)

1. *When consensus meets self-stabilization*, Shlomi Dolev, Ronen Kat, Elad M. Schiller. Journal of Computer and System Sciences, 2010

Transformation systématique

S'il est possible de détecter une faute, on peut transformer un algorithme non autostabilisant en algorithme autostabilisant :

1. Faire tourner l'algorithme original
2. En parallèle : détecter les fautes
3. En cas de faute, appliquer un algorithme autostabilisant de réinitialisation (*reset*) pour revenir à un état légal prédéterminé
4. Arrêter et redémarrer l'algorithme original

(l'arnaque est dans "arrêter l'algorithme original" : nécessité de distinguer les différentes instances ?)

Avantages de l'autostabilisation

- Pas de nécessité d'initialisation du système
- Traitement implicite des fautes transitoires (corruption, perte de message)
- La propriété d'autostabilisation ne dépend pas de la nature de la faute (tant que les sites sont vivants)
- La propriété d'autostabilisation ne dépend pas de l'étendu de la faute (tous les sites !)
- Si une faute est détectée, il suffit de réinitialiser sauvagement les composants concernés
- En absence d'attaquant malveillant, les fautes transitoires sont rares (corruption mémoire, perte de message) $\rightarrow$ le système est presque toujours dans un état légal

Inconvénients de l'autostabilisation

- Convergence à terme $\rightarrow$ perte temporaire de sûreté
- Pas de détection de la stabilisation
- Une faute locale peut impliquer une action sur tous les sites
- Non tolérance aux fautes permanentes (arrêt d'un site)
- Surcoût de l'autostabilisation (en particulier, échange perpétuel de messages)
- Il existe des problèmes sans solution autostabilisante (p.e. sur réseau anonyme ou symétrique)
- Algorithmes complexes, vérification difficile (pas d'invariant contraignant l'espace d'états)

Conclusion

- La coordination (p.e. le consensus) est coûteuse et peu performante
- Certains problèmes peuvent être formulés de manière à se contenter de convergence à terme vers un état valide
- Ils ne nécessitent alors pas de coordination globale tout en tolérant des défaillances partielles : CRDT, problèmes monotones, autostabilisation
- ... mais ce n'est pas possible pour tous les problèmes $\rightarrow$
  - modifier le problème en acceptant des incohérences corrigées par l'humain
  - ou utiliser de la coordination hors du chemin critique ou le plus tard possible
  - ou pas le choix, accepter le coût de la coordination