Fiabilité algorithmique dans les systèmes distribués

Arnaud Labourel

Section 1

Algorithmes pour la Fiabilité

Tâches distribuées fondamentales

Il y a deux tâches très importantes en algorithmique distribuée :

- **Consensus** : se mettre d'accord sur une donnée $\Rightarrow$ tous les processus décident de la même valeur
- **Élection** : faire passer exactement un processus dans un état élu

Ces deux tâches sont très importantes, car elles permettent de résoudre de nombreuses tâches distribuées naturelles.

Tâche distribuée : consensus

Le consensus est *approximativement* la tâche que résout, de manière répétée, la *blockchain*.

$\Rightarrow$ répéter le consensus permet de maintenir une structure de données cohérente qui est dupliqué sur plusieurs serveurs.

Objectif

Construire un système distribué de serveurs qui se comportent comme un seul serveur cohérent du point de vue des clients.

$\Rightarrow$ pas toujours possible en fonctions des pannes

Accord Distribué

Les tâches distribuées, dans ce contexte, se ramènent en général à une forme d'*accord distribué*.

La plus importante : le **Consensus**.

Observation Tous les nœuds exécutent le même algorithme. Seul l'état initial et/ou l'identifiant peuvent différer.

Motivations du Problème du Consensus

Réplication de bases de données distribuées :

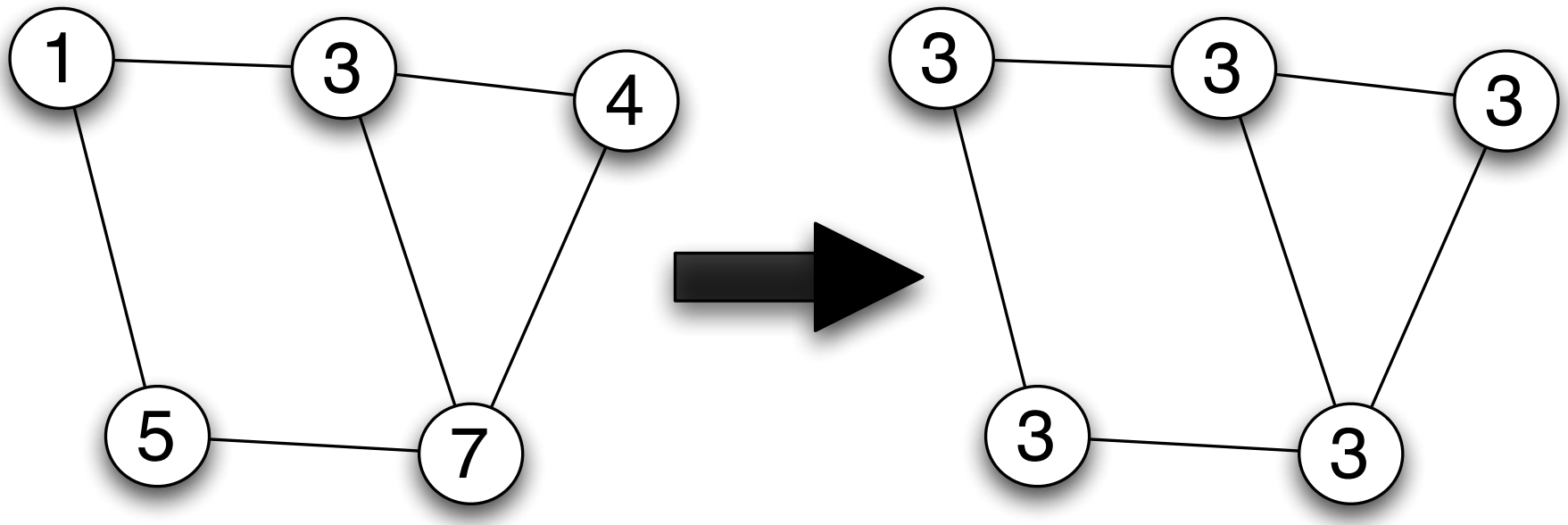
- comment les rendre **identiques** après une panne, une interruption réseau ...

Réplication de machine à états :

- déplacer une machine virtuelle d'un serveur à un autre

Intuitivement

Les processus choisissent une, et une seule, valeur parmi celles initialement présentes



Problème du Consensus

Décider : On dit qu'un processus *décide* lorsqu'une valeur **finale** pour le calcul est donnée par celui-ci.

Spécification

Étant donnés des processus avec chacun une valeur initiale dans Λ,

Accord	Tous les processus qui décident une valeur décident la même valeur.
Intégrité	La valeur décidée est une valeur initialement proposée
Terminaison	Tous les processus **non défaillants** décident.

Paxos

Algorithme de L. Lamport *The Part-Time Parliament* ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169.

SRC Research Report 49 (1990)

famille d'algorithmes nombreux articles et algorithmes :

Vertical Paxos and Primary-Backup Replication (Leslie Lamport, Dahlia Malkhi and Lidong Zhou) Proceedings of PODC 2009, pp312-313.

Que Résout l'Algorithme ?

Dans un système où les processus

- peuvent communiquer par messages les uns avec les autres,
- mais certains messages peuvent se perdre ou mettre du temps à arriver
- et un processus peut crasher sans prévenir

maintient un état global cohérent.

Attention

Le problème est beaucoup plus difficile que ce que laisse penser sa description.

Raft : alternative à Paxos

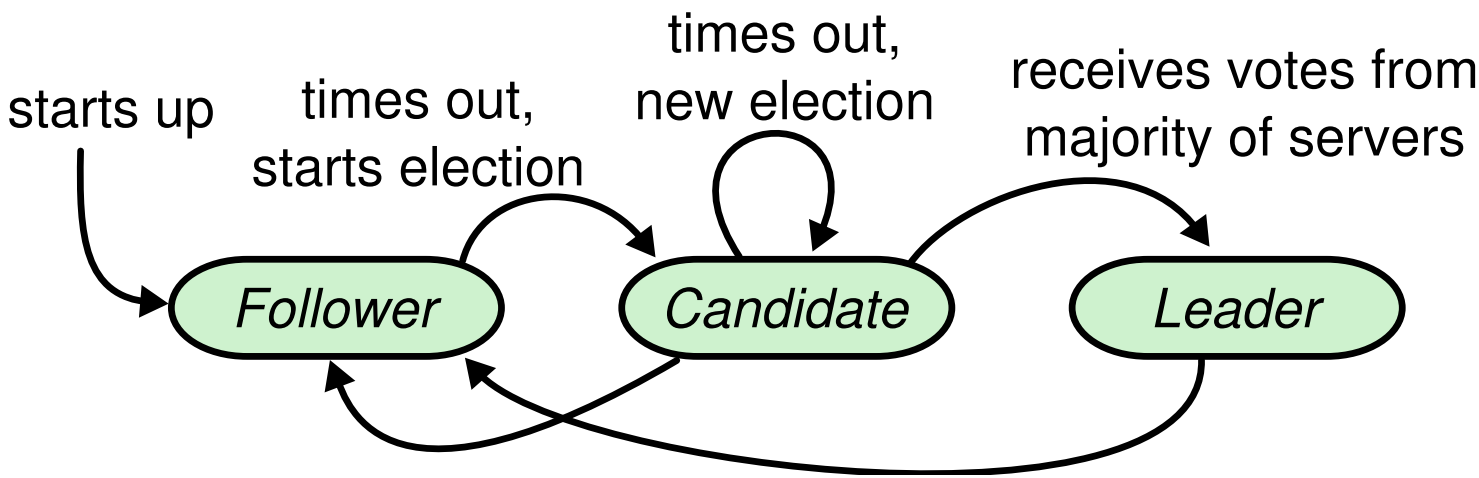
Paxos est complexe (quelques explications dans la suite du cours) et très compliqué à implémenter.

$\Rightarrow$ D'autres algorithmes de consensus "plus simple" ont été proposé comme Raft.

Diego Ongaro and John K. Ousterhout, *In Search of an Understandable Consensus Algorithm.* USENIX Annual Technical Conference 2014: 305-319

Raft : principes clés

Trois états possibles pour les serveurs : `follower`, `leader` et `candidate`.



- utilisation de *timers* aléatoire pour casser les symétries (élection)
- dès qu'un serveur est candidat il envoie des messages à tous les autres processus pour qu'il vote pour lui
- une fois élu le `leader` prends toutes les décisions et envoie des messages régulièrement à tous les autres serveurs

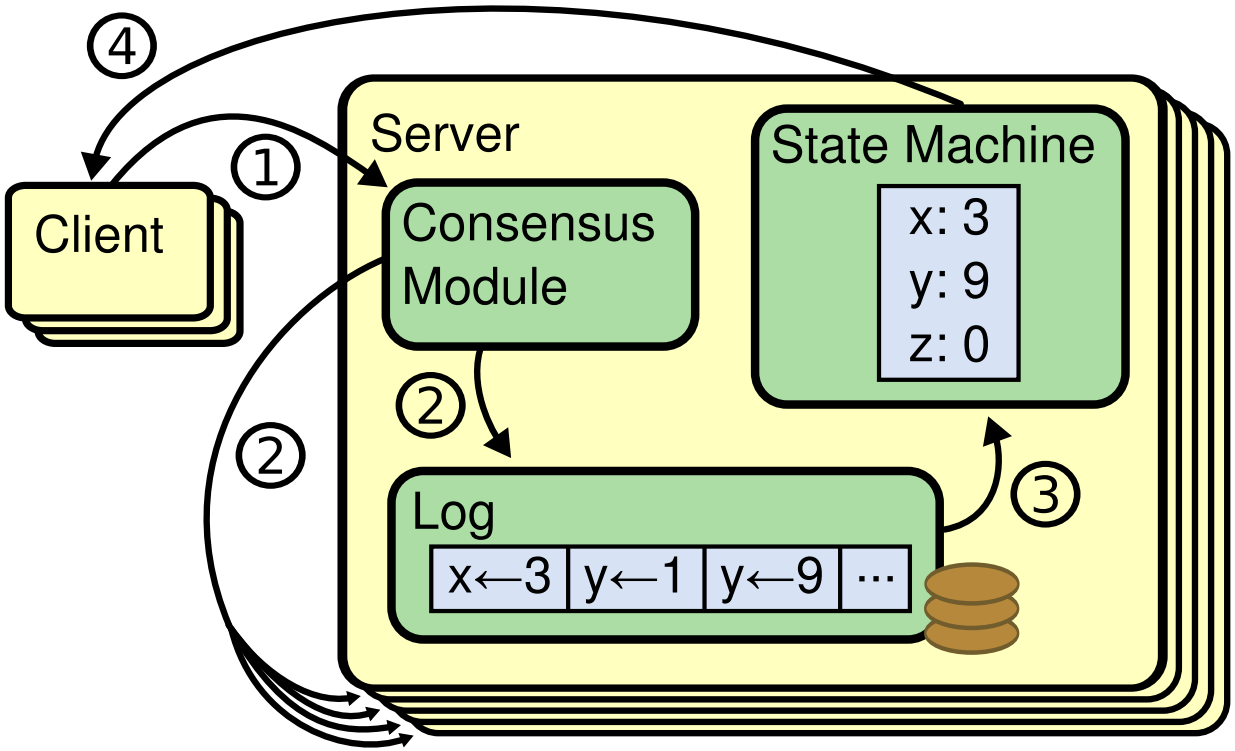
Section 2

Consensus en quelques exemples

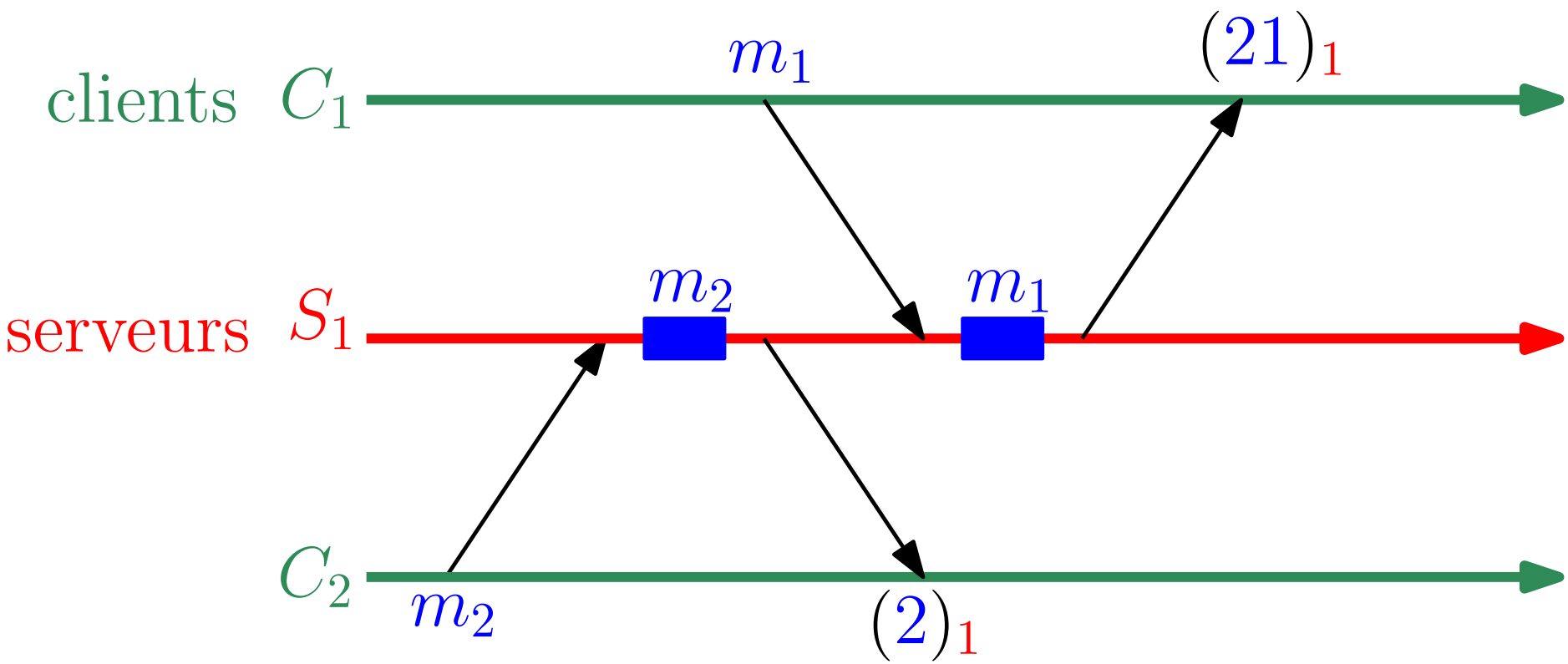
Application du consensus

Une des principales applications du consensus est de maintenir une structure de données cohérente entre plusieurs serveurs.

Une façon de réaliser cela est de se mettre d'accord sur un ordre total des requêtes reçues par les serveurs.



Un seul serveur

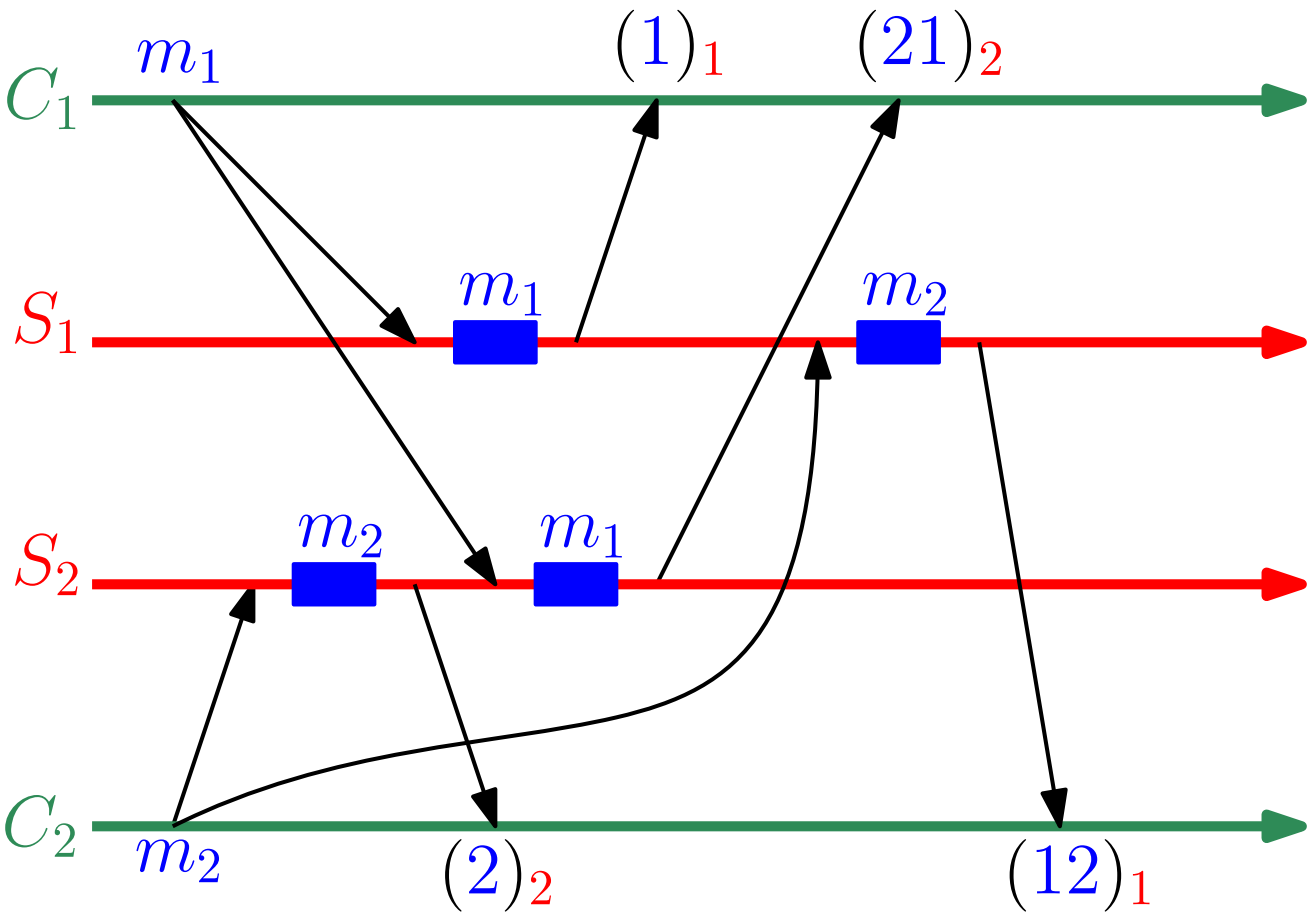


$(abc)_i$: état du serveur i ayant reçu des requêtes m_a, m_b et m_c dans cet ordre

Remarque

Cas très facile.

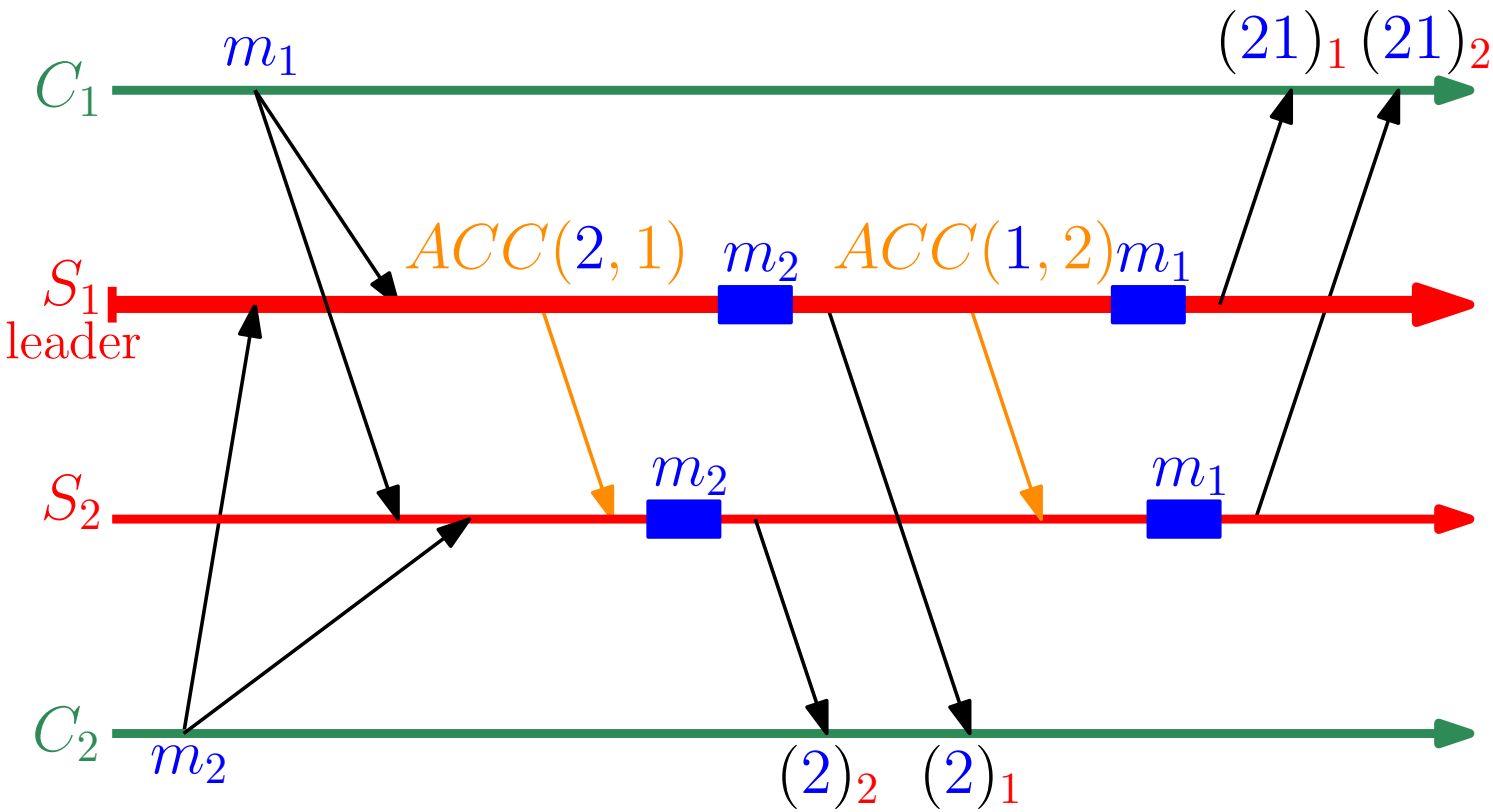
Deux serveurs



Remarque

Pas d'accord sur l'ordre des requêtes sans communication entre les serveurs.

Deux serveurs dont un leader



$ACC(i,j)$: message demandant l'acceptation de m_i en tant que j-ème requête

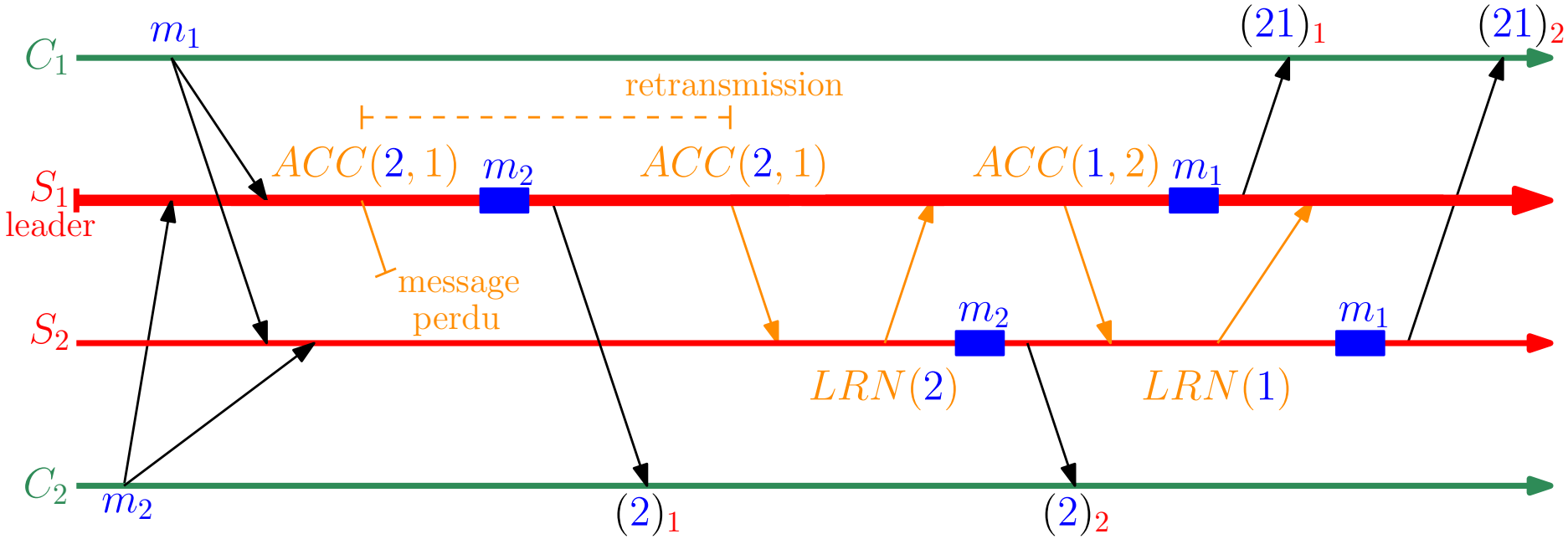
Remarque

Fonctionne s'il n'y a pas de pertes de messages.

Deux serveurs avec pertes de messages

Nouvelle règle

Ajout d'accusé de réception et mécanisme de retransmission

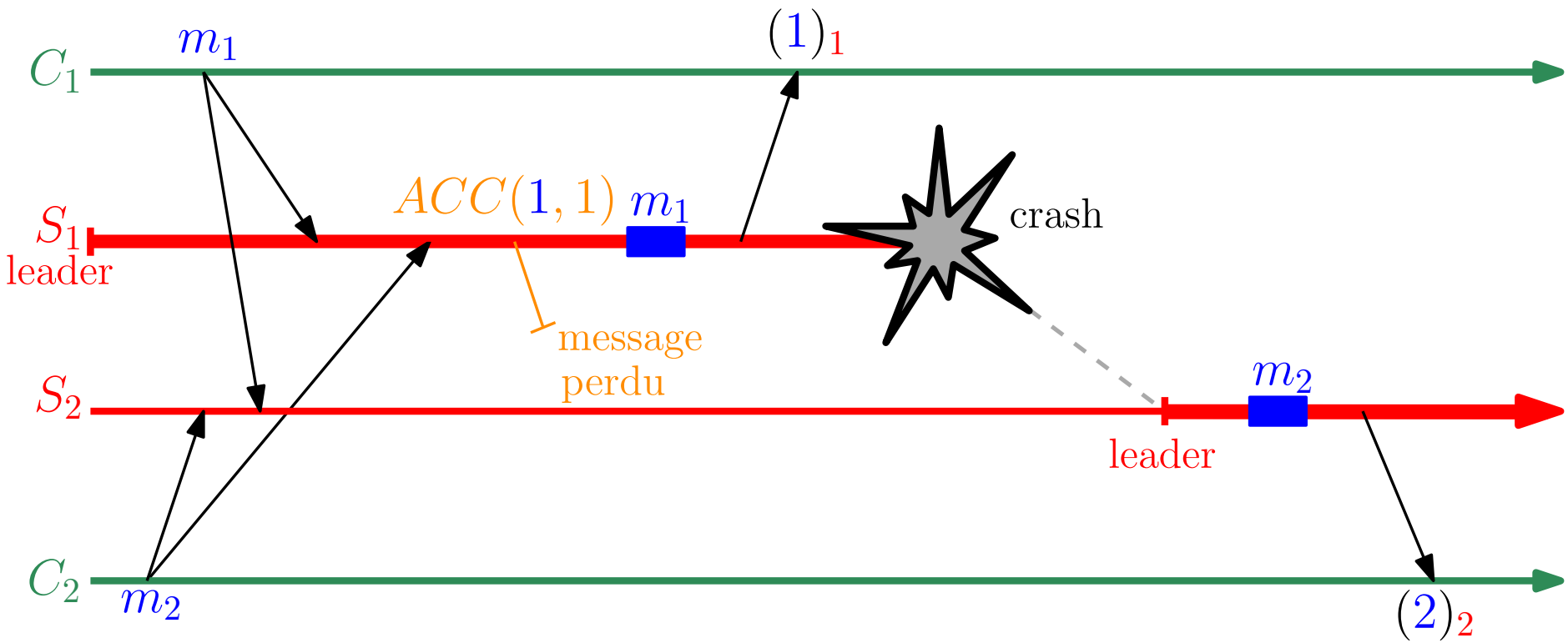


$LRN(i)$: message accusant l'acceptation d'un message $ACC(i,j)$

Deux serveurs avec crash

Nouvelle règle

Le serveur non-*leader* exécute l'algorithme pour un serveur s'il n'a plus de messages de la part du *leader* pendant un certain temps.

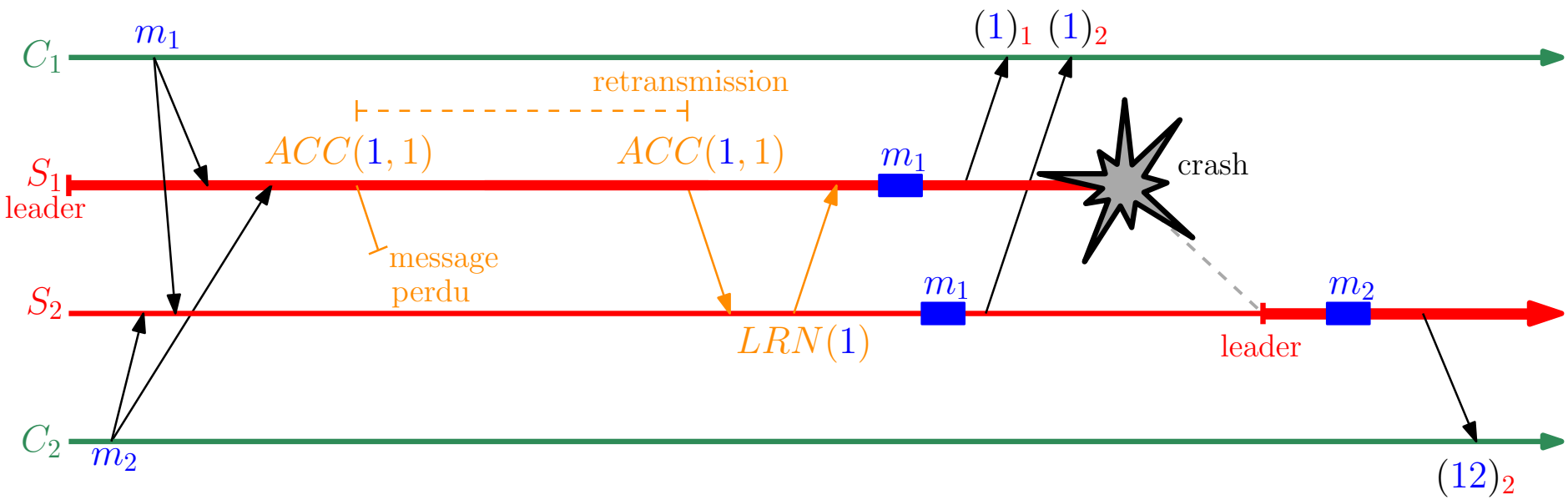


$\Rightarrow$ perte de la cohérence de l'ordre des requêtes

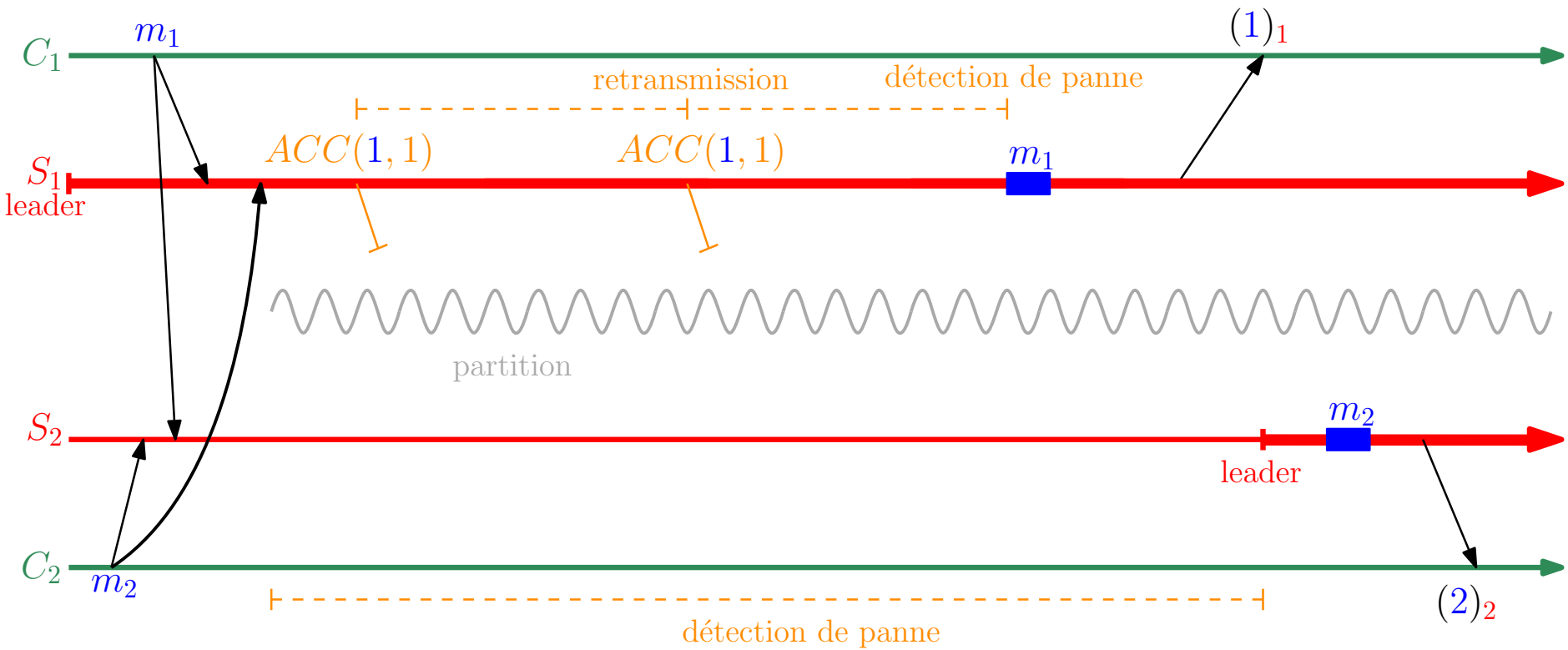
Deux serveurs avec crash

Nouvelle règle

Le serveur *leader* attends l'accusé de réception de l'autre serveur avant de traiter une requête.



Partition du réseau



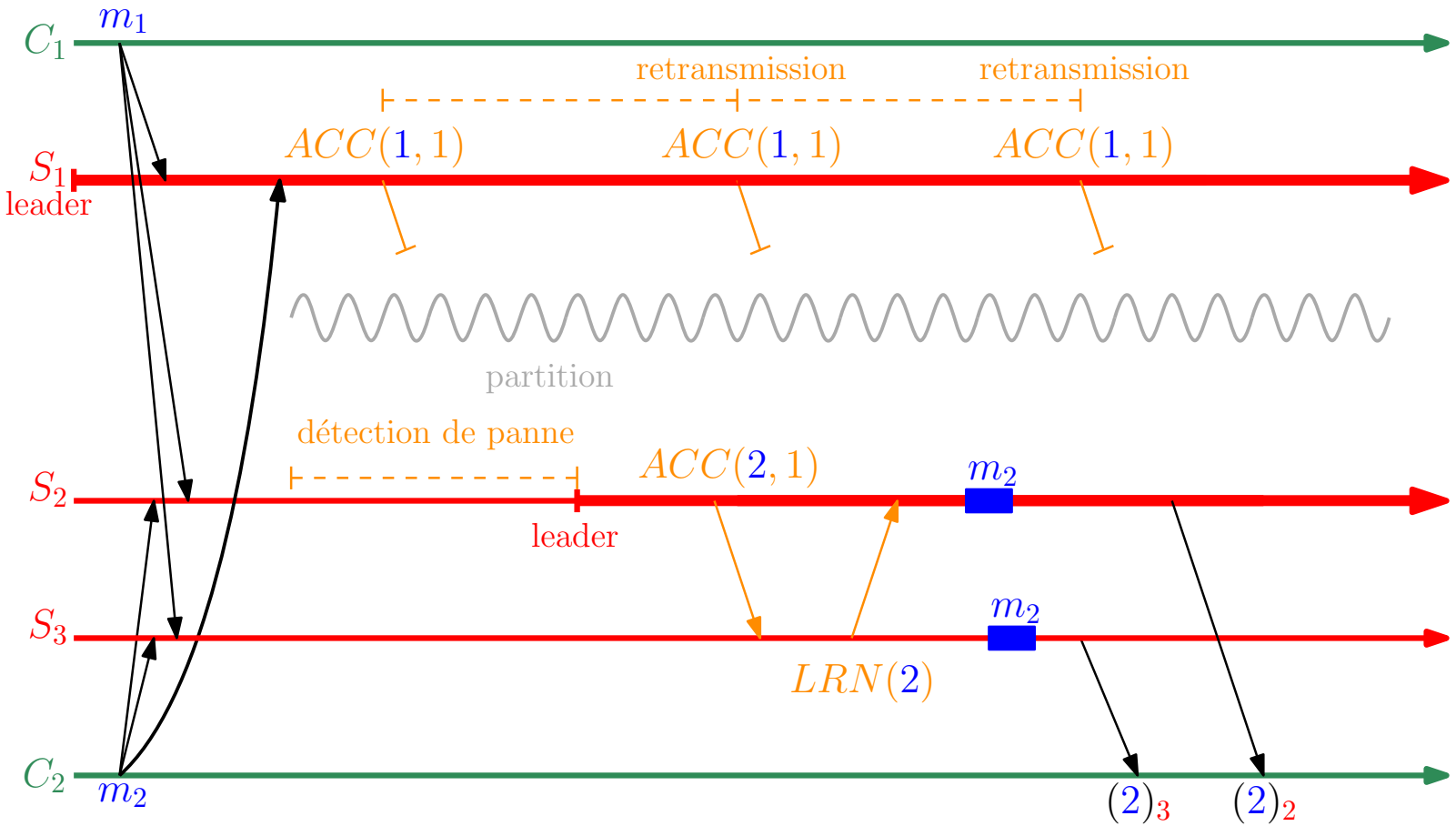
Problème

En cas de partition du réseau il peut y avoir 2 *leaders* et des décisions incohérentes.

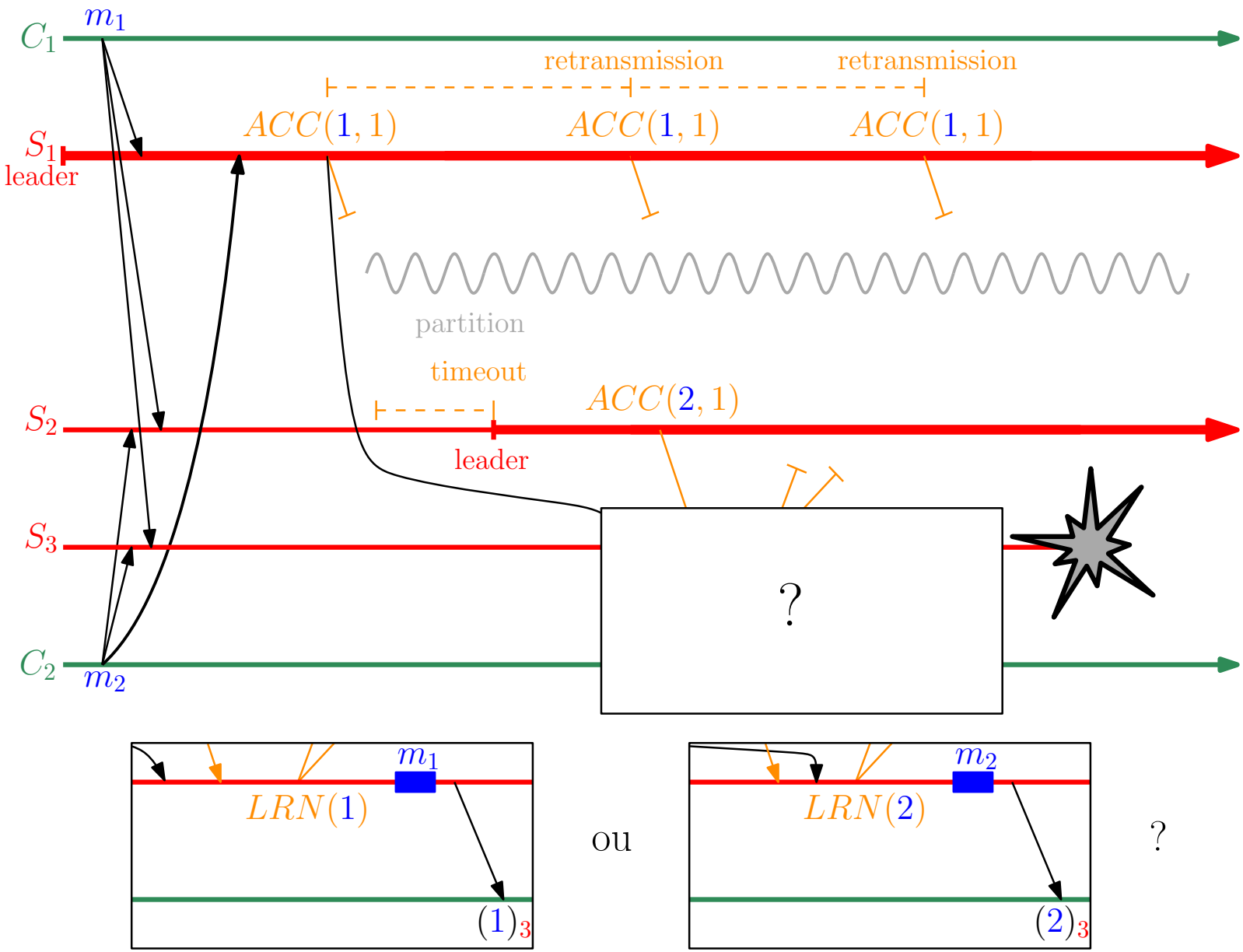
Partition du réseau à 3

Nouvelle règle

Une partie avec une majorité de processus a le droit de progresser.



Incertitude en cas de partition puis crash



Mécanisme de passation de pouvoir

Difficulté

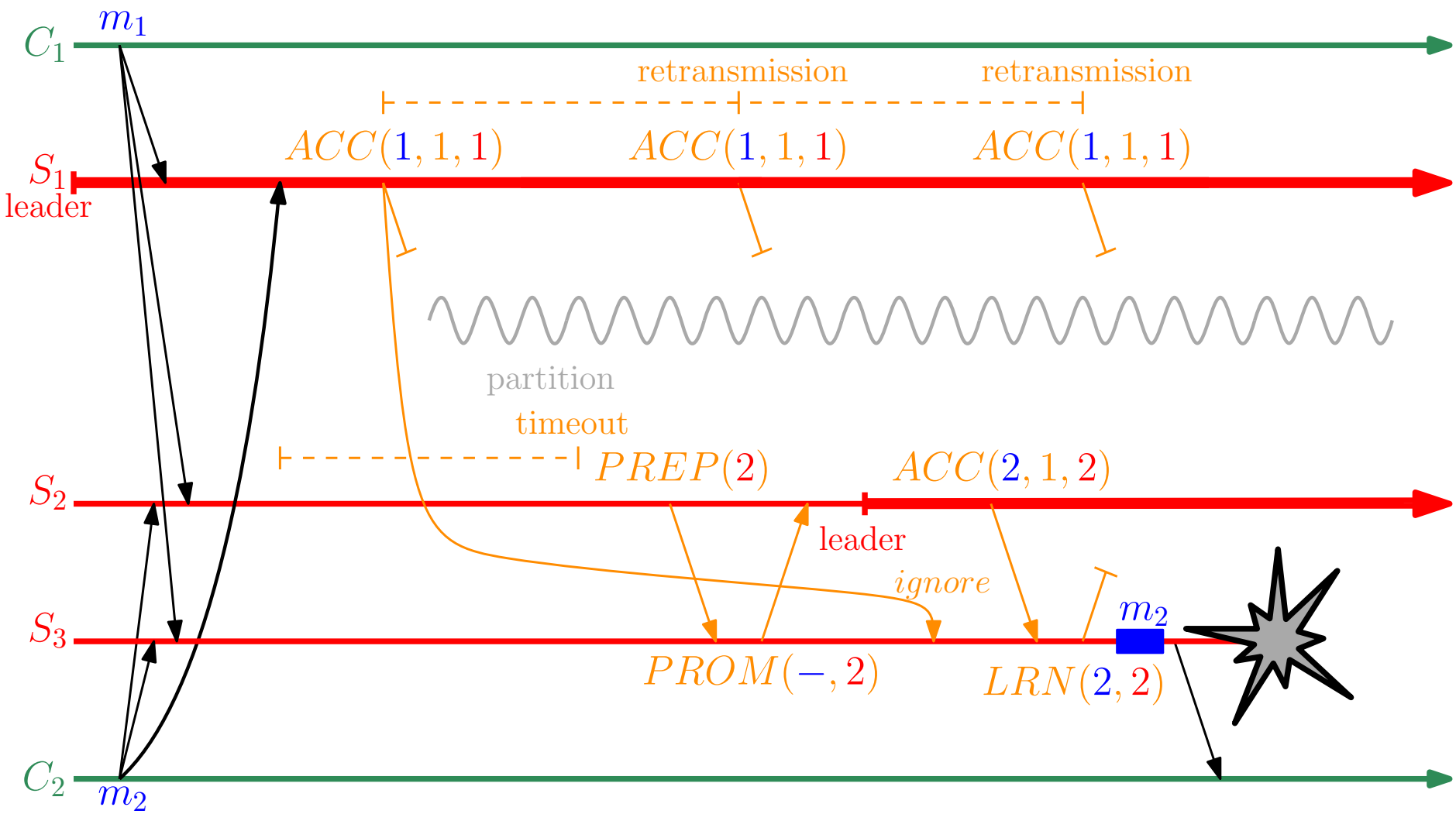
Plusieurs *leaders* peuvent envoyer des messages d'acceptation.

Solution

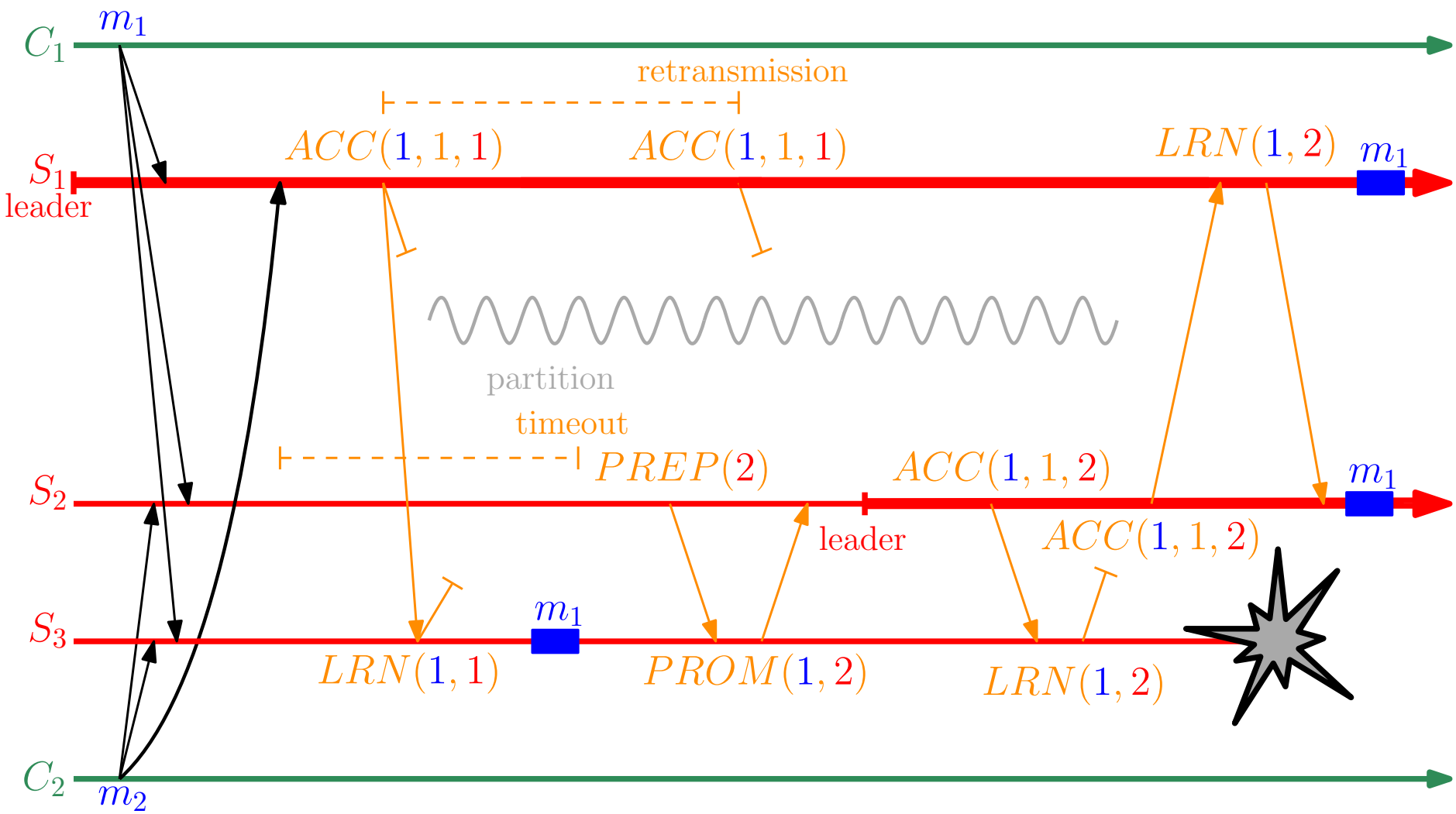
Mécanisme de prise de pouvoir :

- Un processus envoie des messages *PREPare* pour devenir *leader*
- Il attend d'avoir reçu une majorité d'accusé de réception sous forme de *PROmise* pour devenir *leader*
- Les messages *PROmise* contiennent la liste des requêtes déjà traitées par le serveur
- Les messages contiennent le numéro du serveur afin de pouvoir ignorer ceux issus d'un ancien *leader* qui s'est fait remplacé.

Mécanisme de passation de pouvoir



Mécanisme de passation de pouvoir



Cinq serveurs

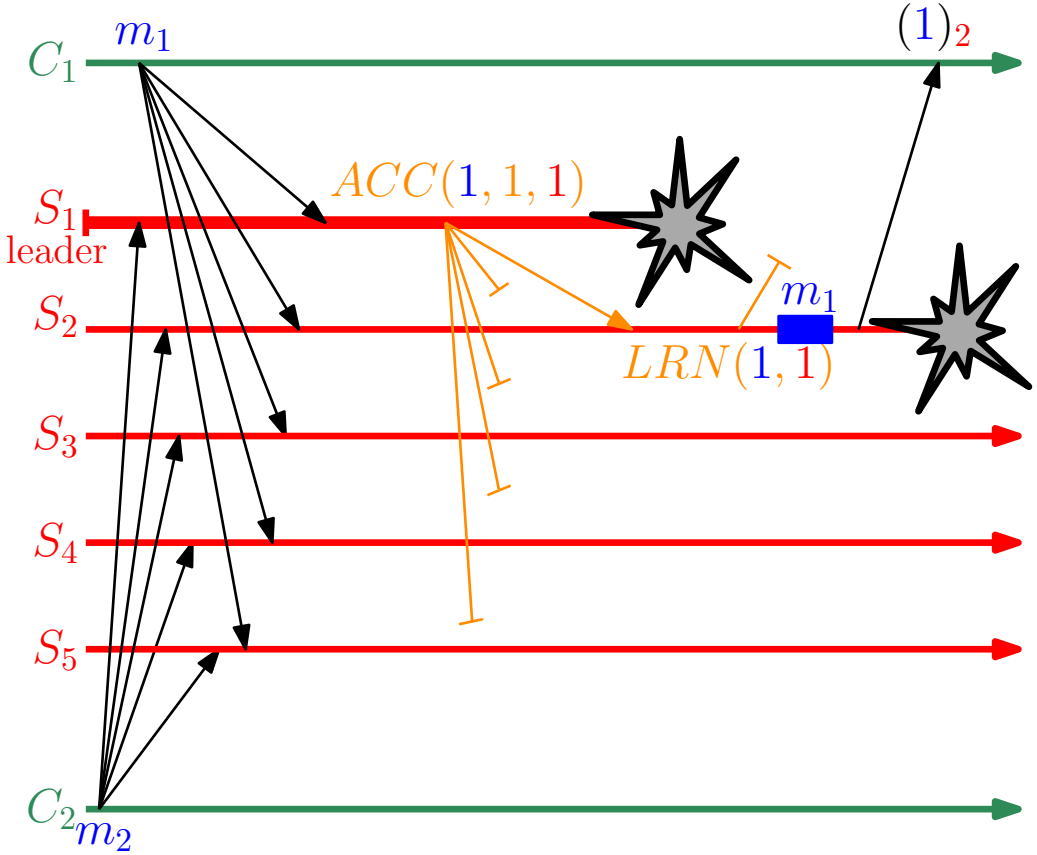
Pour tolérer f pannes, il faut $2f + 1$ serveurs

Règles

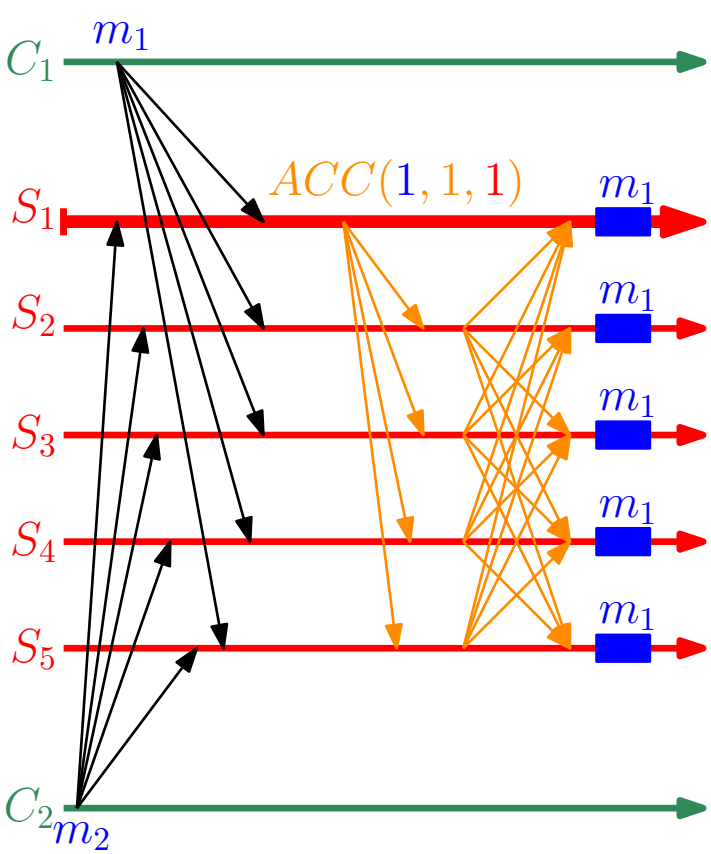
- Traiter les requêtes après réception de $f + 1$ *LRN* :
  - les *ACC* du *leader* sont considérés comme des *LRN*
  - un serveur s'envoie automatiquement des *LRN* à lui-même
- Devenir *leader* après avoir reçu f *PROM* (majorité avec le serveur lui-même)

$\Rightarrow$ les serveurs envoient les messages à tous les autres serveurs.

Cinq serveurs



risque de perte d'information si on traite une requête après réception d'un seul ACC



Traiter les requêtes après réception :
- d'1 ACC et d1 $PROP$, ou
- de 2 $PROP$

Raft

Différence avec Paxos et autres

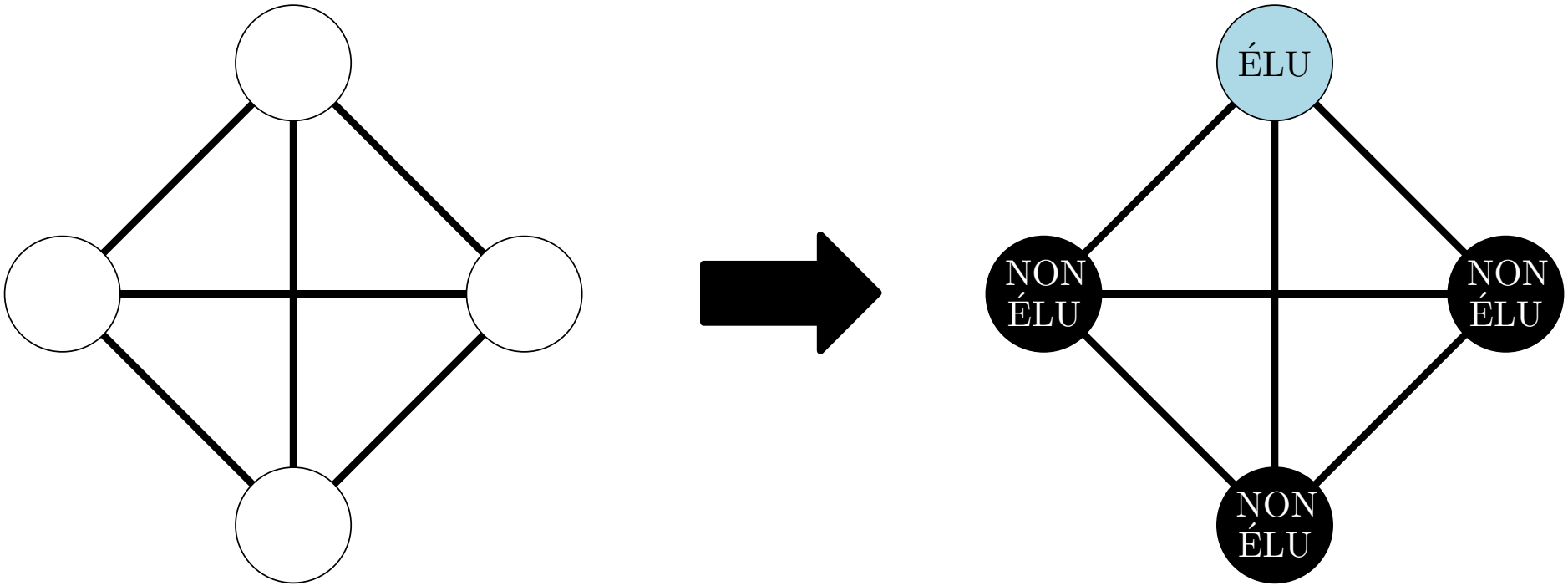
Élection forte plus forte que la plupart des algorithmes, par exemple les logs ne vont que dans le sens du leader vers les autres serveurs.

Algorithme d'élection Raft utilise des temporisations aléatoires pour élire des leaders, cela n'est guère plus lourd que les **battements de cœur** (*timeout*) tout en résolvant les conflits plus facilement.

Changement des membres Le mécanisme pour changer les serveurs utilise une nouvelle approche qui permet de faire se superposer les systèmes pendant la transition, ce qui évite l'interruption de service.

Parenthèse sur l'Élection

Intuitivement, l'élection revient à choisir un *leader* parmi les nœuds du système.



Parenthèse sur l'élection

Étant donné des processus,

Élection

- un processus et un seul, doit terminer dans l'état ÉLU.

Variante

en outre

- les autres processus doivent terminer dans l'état NON_ÉLU.

Remarques (particularités du distribué)

- pas d'entrée
- plusieurs types de terminaisons possibles

Élection dans un arbre

C'est le cas le plus simple : algorithme d'effeuillage

Règles

- un nœud ayant reçu un message **`devient mon père`** de tous ces voisins sauf au plus un, les reconnaît comme fils et leur envoie **`je suis ton père`**.
- si un nœud est devenu père de tous ces voisins sauf un, il envoie un message **`devient mon père`** à son voisin restant.

Problème possible sur les deux derniers sommets $\Rightarrow$ pas de solution déterministe en général (symétrie)

norme IEEE 1394

Section 3

Fondamentaux pour les données réparties

Difficultés

Algorithmes de Consensus et bases de données

Paxos et Raft permettent de maintenir une base de données répartie et cohérente
$\Rightarrow$ plus exactement *jamais incohérente*

Toutes les propriétés souhaitables ne sont pas forcément atteignables en même temps !

Propriétés Souhaitables ?

Malgré les problèmes pratiques incontournables, on souhaite avoir des garanties de sûreté et de progrès.
Mais lesquelles ?

Propriétés ACID

Atomicité

Une transaction se fait au complet ou pas du tout

Cohérence

Chaque transaction est une transition d'un état valide à un autre état valide.

Isolation

Toute transaction doit s'exécuter comme si elle était la seule sur le système.

Durabilité

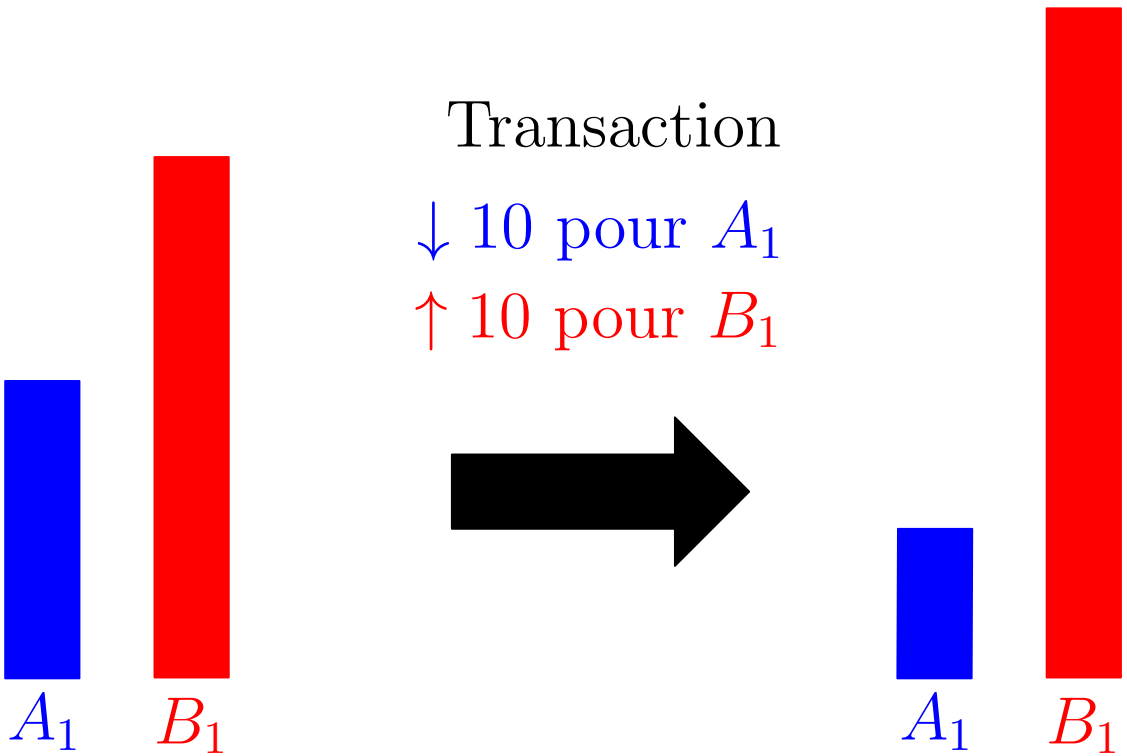
Une transaction validée est définitive

Exemple ACID sur une BD

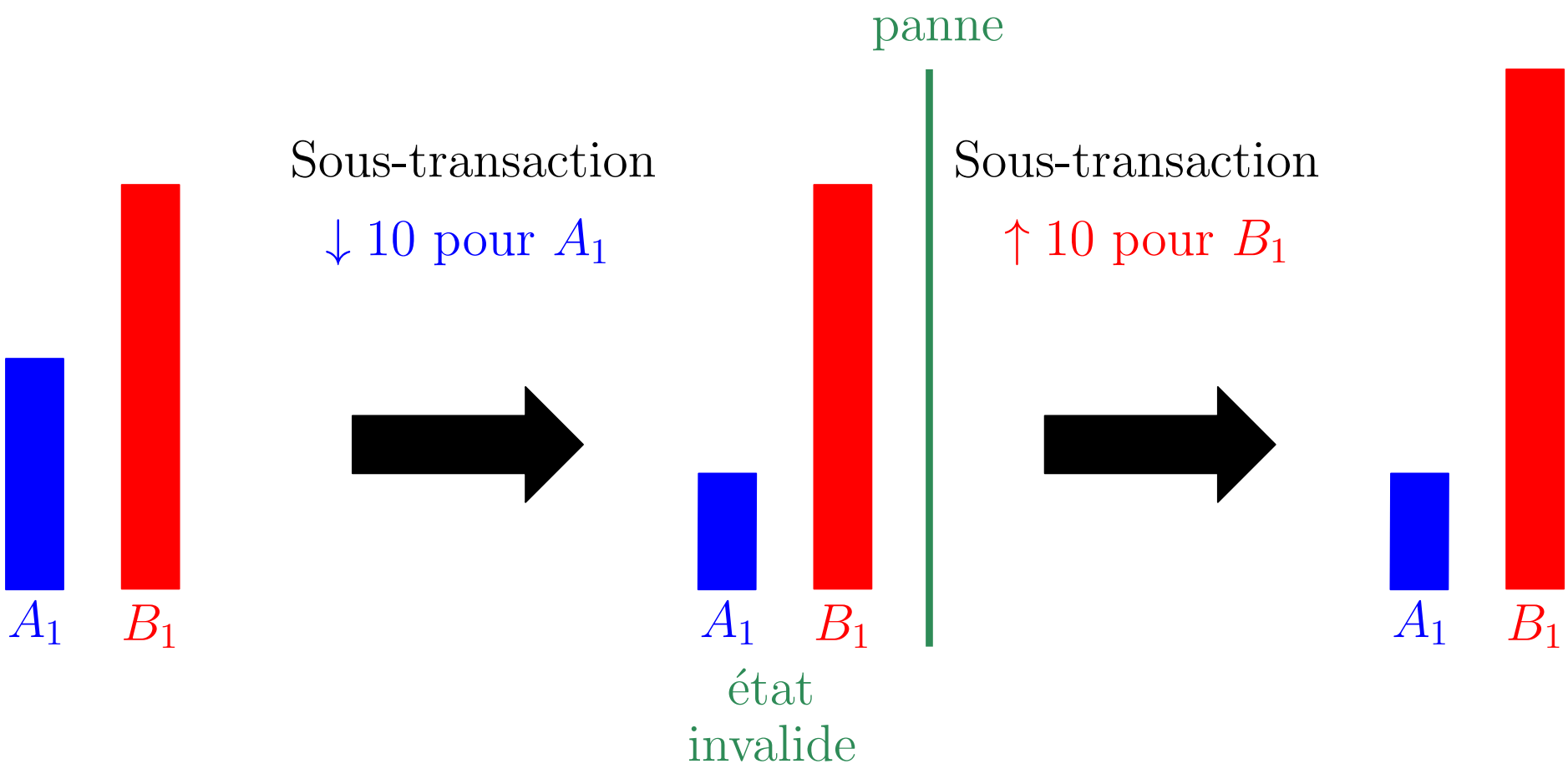
Table ayant deux champs dont la somme doit valoir 100 :

```
CREATE TABLE t (A INTEGER, B INTEGER CHECK (A+B = 100));
```

Exemple de transaction :



Exemple **A**CID sur une BD : Atomicité



$\Rightarrow$ pour respecter l'**atomicité** il faut que les sous-transactions soient annulées en cas de panne afin que la base retrouve son état initial.

Sauvegarde des valeurs avant transaction ?

Exemple A**C**ID sur une BD : Cohérence

Pour garantir la **cohérence** il faut que les transactions menant à un état invalide soient aussi annulées.

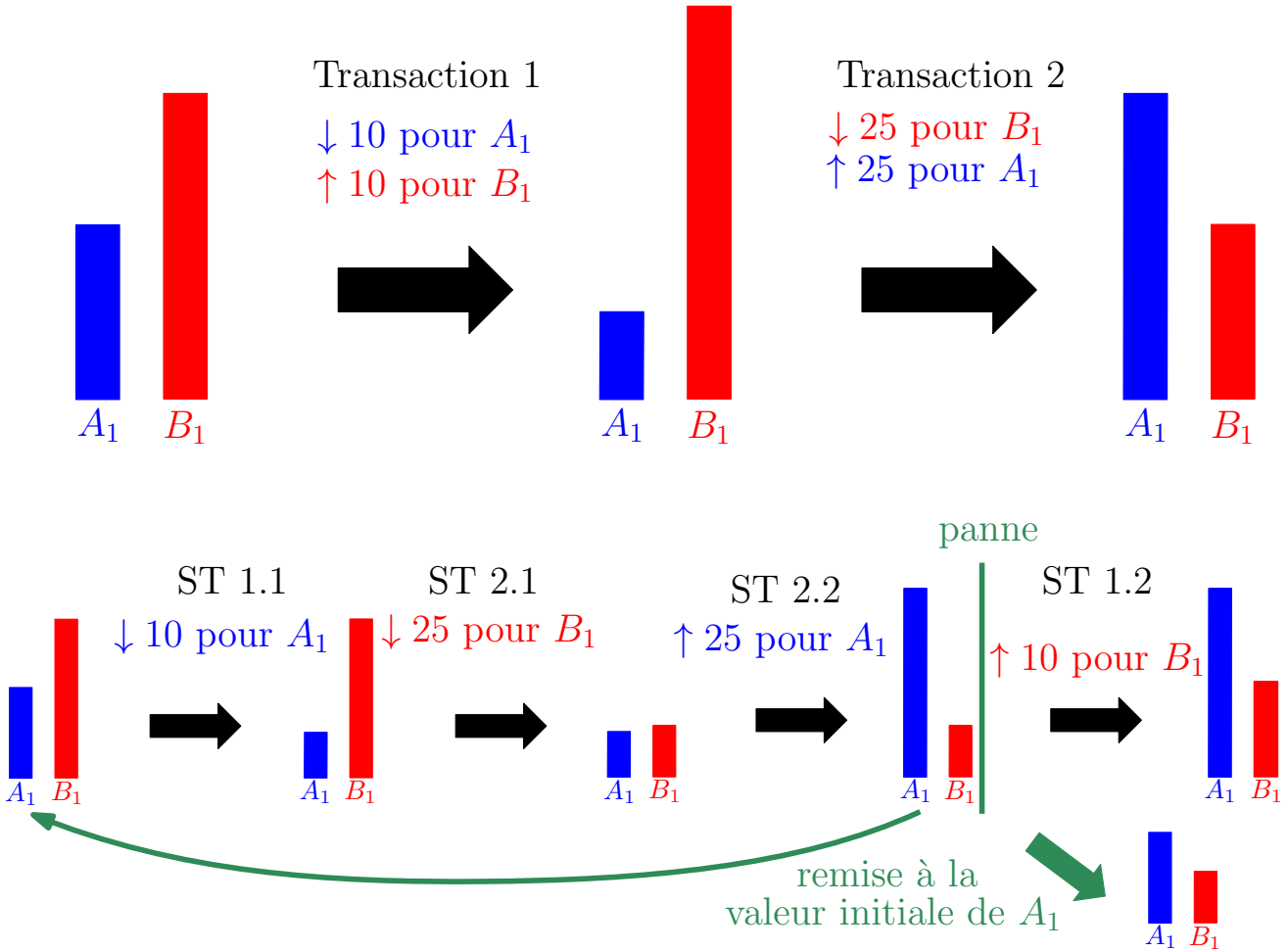
Procédure :

1. faire la transaction
2. vérifier la validité de l'état du système
3. annuler l'effet de la transaction au besoin.

Nécessité de pour voir revenir en arrière

rollback : opération remettant une BD à un état précédant le début d'exécution d'une transaction

Exemple ACID sur une BD : Isolation



$\Rightarrow$ le système doit être muni de contrôles et de rollbacks qui assurent que, dans tous les cas, le résultat de l'exécution simultanée des transactions donne le même résultat que leur exécution en séquence.

Exemple ACI**D** sur une BD : Durabilité

Exemple d'échec de durabilité

1. Une transaction est validée dans la mémoire tampon *buffer* du disque.
2. Une validation est envoyée à l'utilisateur.
3. Une panne d'électricité survient avant que la transaction soit effective sur le disque.

$\Rightarrow$ compromis à faire entre le temps de réponse et la garantie que la requête est validée

Cadre Général Distribué

Tolérance aux pannes

Ces propriétés doivent être garanties même après :

- panne d'électricité,
- crash d'un des serveurs,
- pertes de certains messages,
- coupure du réseaux.

Impossibilités

Réaliser le Consensus robuste permet de garantir ces propriétés.
Cependant, plusieurs résultats d'**impossibilité** ont été prouvés :

- problème des deux généraux (Gray 1978)
- impossibilité du consensus asynchrone (FLP 85)

Théorème CAP (Brewer 2000; Gilbert-Lynch 2002)

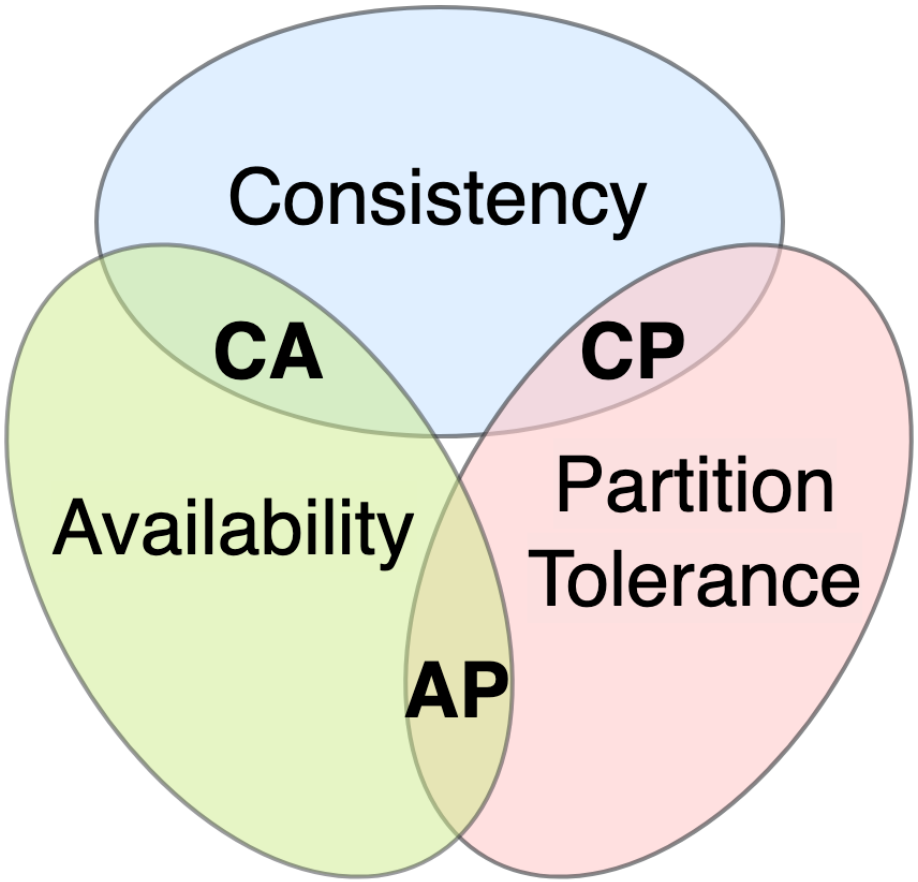
Il n'est pas possible de satisfaire simultanément les trois propriétés :

- **Cohérence (*Consistancy*)** : tous les nœuds du système voient exactement les mêmes données au même moment
- **Disponibilité (*Availability*)** : garantie que toutes les requêtes reçoivent une réponse
- **Tolérance au partitionnement (*Partition Tolerance*)** : Aucune panne moins importante qu'une coupure totale du réseau ne doit empêcher le système de répondre correctement.

Tolérance au partitionnement (définition alternative)

En cas de morcellement en sous-réseaux, chaque sous-réseau doit pouvoir fonctionner de manière autonome.

CAP en un dessin



On peut seulement avoir deux des propriétés de CAP.

Compromis

Comme il est en général impossible de garantir l'absence de partitionnement (mais ça peut être rare), on a en général le choix de privilégier

Consistency SQL : système de BD classique (respect des propriétés ACID)

Avaibility BD `NoSQL` : remise en cause d'ACID afin d'avoir un système extensible (*scalable*) $\Rightarrow$ système de BD distribué permettant de traiter des volumes de données importants.

BASE : l'opposé d'ACID

- **Basically Available :** les opérations de lecture et d'écriture sont disponibles autant que possible mais peuvent être incohérentes (échec possible).
- **Soft-state :** L'état du système peut changer au fil du temps. Même pendant les périodes sans entrée, il peut y avoir des changements dus à la EC.
- **Eventually consistent (EC) :** Le système deviendra finalement cohérent lorsqu'il cessera de recevoir des données. Les données se propageront partout où elles devraient tôt ou tard, mais le système continuera à recevoir des données et ne vérifiera pas la cohérence de chaque transaction avant de passer à la suivante.

PACELC

Il y a également un compromis Consistency/Latency même lorsqu'il n'y a pas de partitions.

PACELC

```
if Partition then
    choose either Availability or Consistency
Else
    choose either Latency or Consistency
```

Wojciech Golab, "Proving PACELC", ACM SIGACT News, Volume 49 Issue 1 (2018)

Section 4

Systèmes décentralisés robustes

Les systèmes fortement décentralisés existent

- Internet
- Blockchain
- ???

Conditions d'existence

Au delà des difficultés algorithmes, et systèmes, un objet technique dépend de conditions socio-économiques.
Le modèle économique importe.

La première blockchain (décentralisée) : Bitcoin

- Monnaie virtuelle avec registre de transactions sur une chaîne de blocs créée en 2008 par *Satoshi Nakamoto*
- Résout le problème du consensus (itéré) en mêlant algorithmique distribuée, cryptographie et incitations monétaires

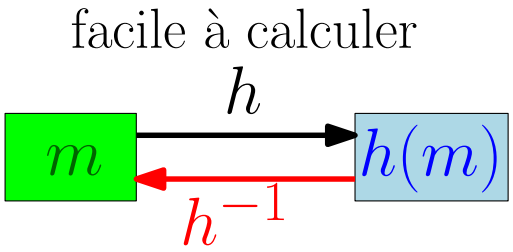
Technologies de base pour le Bitcoin

- Paire clés publique/privé pour authentifier une transaction
- Chaîne de blocs
- Arbres de Merkle
- Une manière de résoudre le consensus :
  - Preuve de travail : loterie utilisant des condensats (*hash*) cryptographiques pour choisir le prochain bloc
  - Preuve d'enjeu : élection entre comptes ayant mis en gage un enjeu afin de déterminer qui choisit le prochain bloc

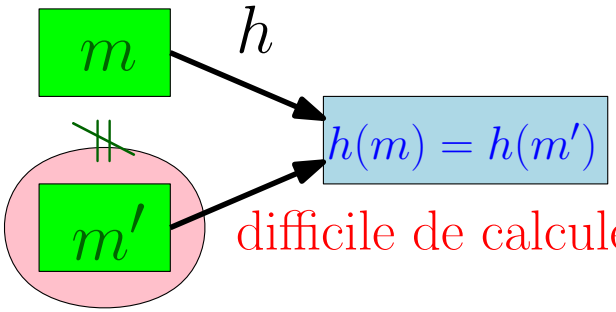
Interrelation algorithmique / économique

Il n'y a pas d'intérêt économique à faire n'importe quoi sur la chaîne.

Rappel fonction hash cryptographique

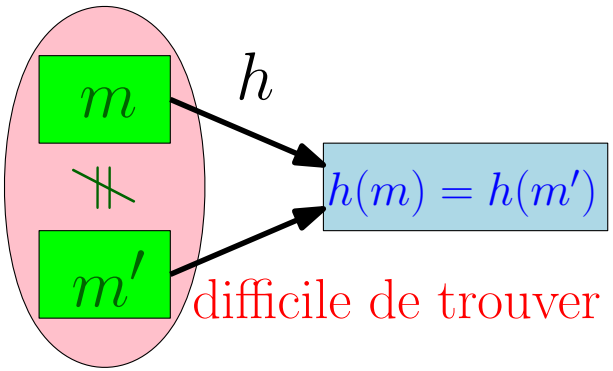


Pre-image resistance



Second pre-image resistance

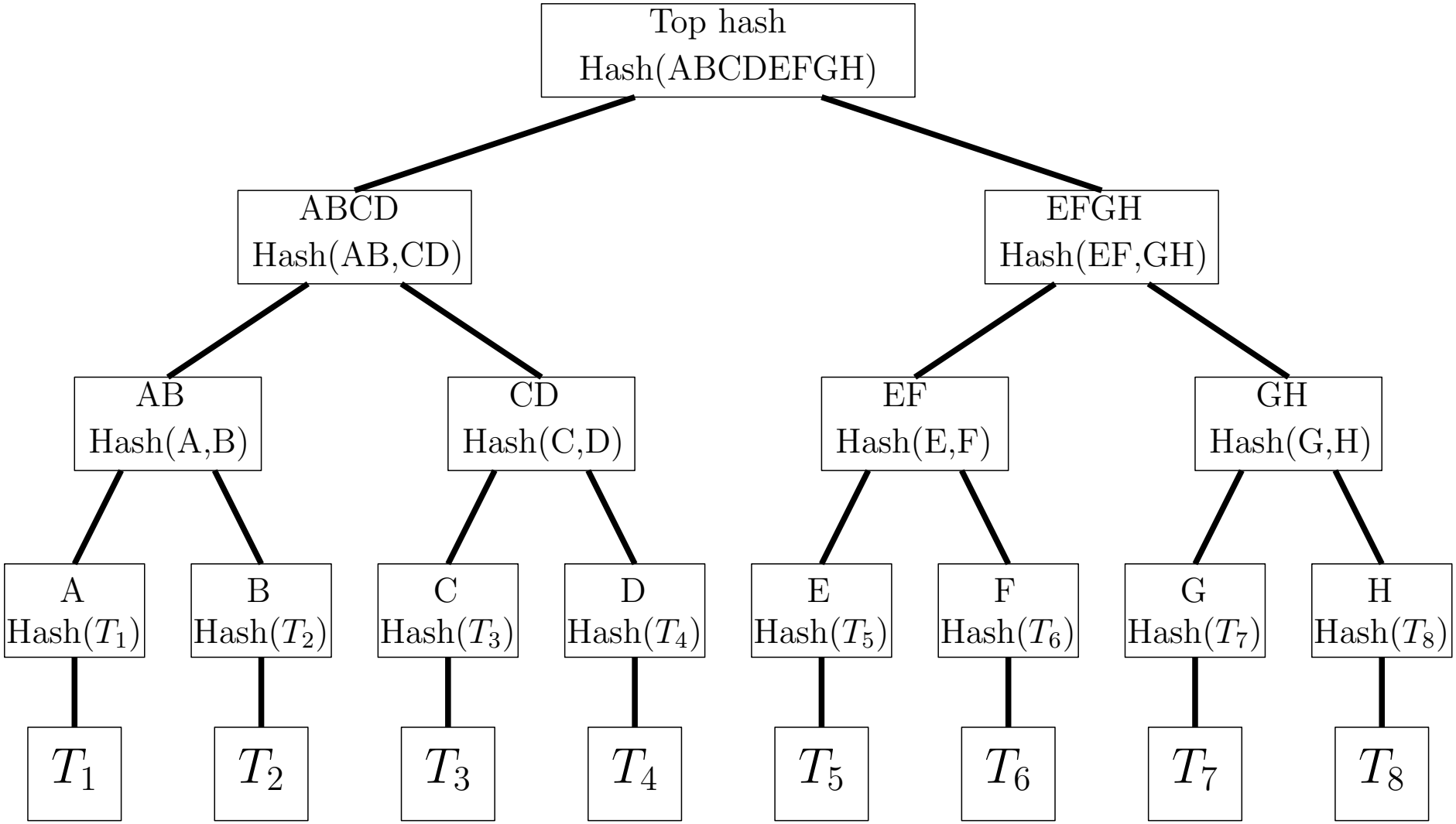
difficile de calculer m' tel que $m' \neq m$



Collision resistance

difficile de trouver une paire (m, m') telle que $m' \neq m$

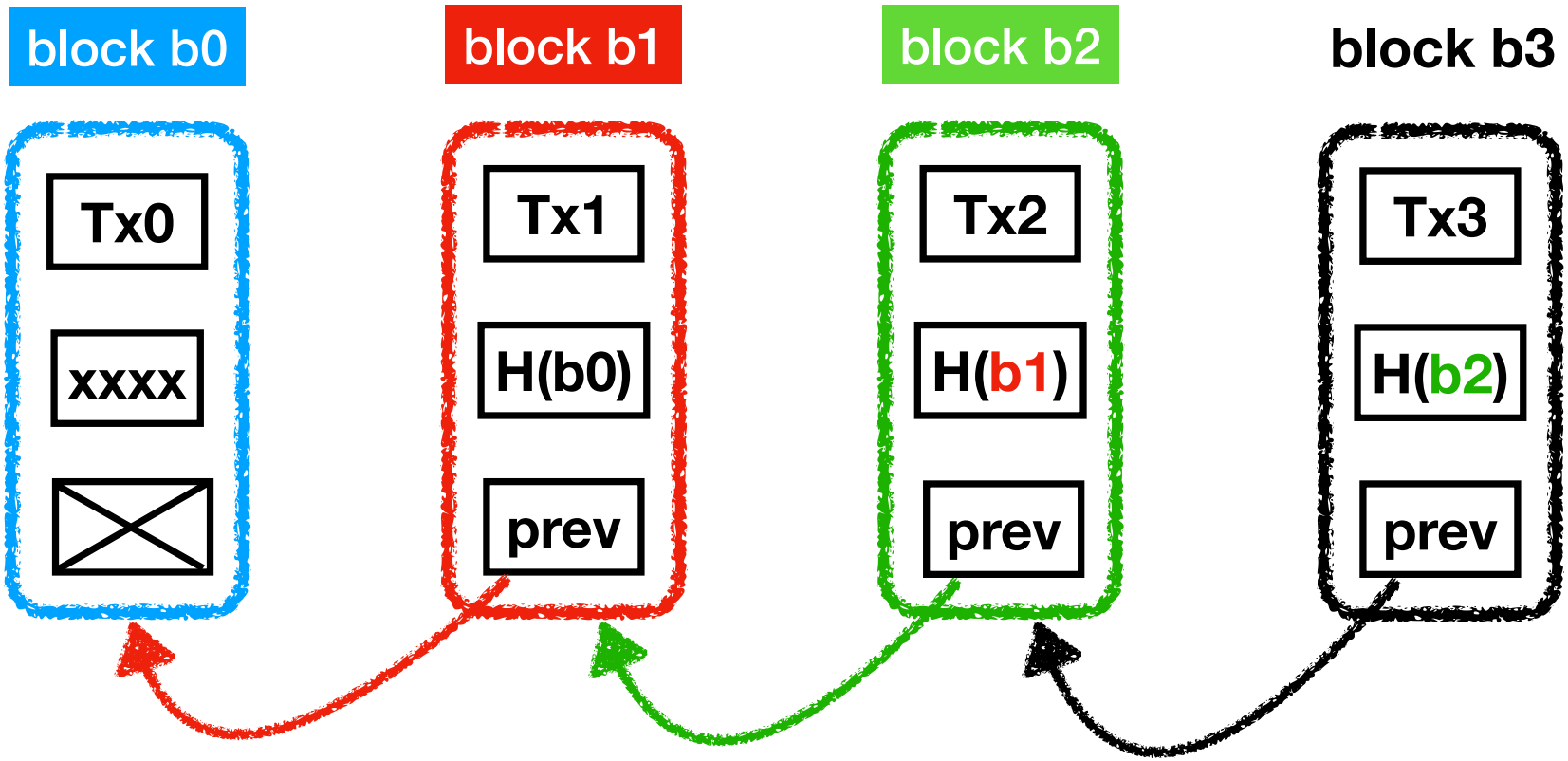
Arbre de Merkle



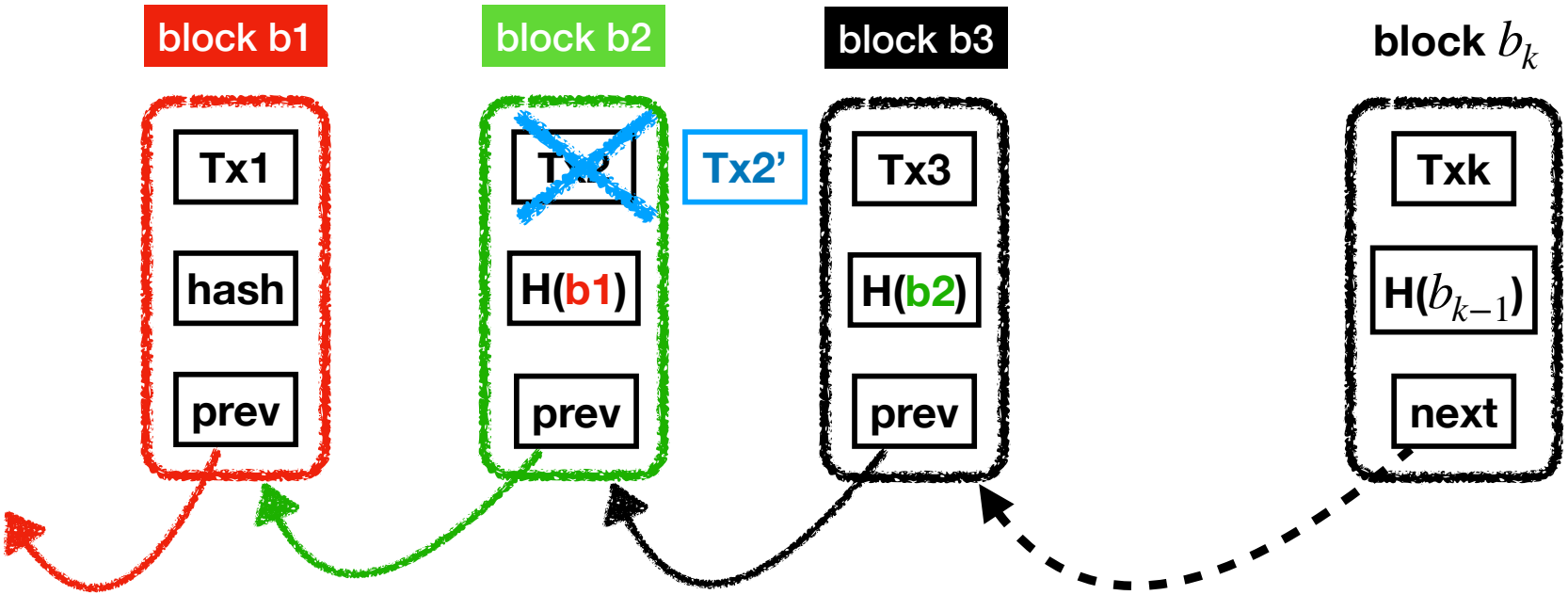
- Chaque feuille est étiquetée avec le hash d'une transaction
- Chaque nœud interne est étiqueté avec le hash de ses enfants

Chaîne de blocs

- L'ensemble de toutes transactions est stocké dans une chaîne de blocs
- Chaque bloc contient des transactions et un hash du bloc précédent



Chaîne de blocs



Pour remplacer Tx2 $\rightarrow$ $Tx2'$, il faut

- modifier b2 en b2' tel que H(b2) = H(b2') (difficile)
- ou modifier les hash dans b3, ... , bk (difficile)

Problème de consensus dans un système ouvert

Pour le Bitcoin, les nœuds qui peuvent être byzantin doivent se mettre d'accord sur les transactions à valider.

Rappel : byzantin = sur-approximation de "ne respecte pas le protocole"

Théorème [Lamport, Shostak, Pease]

Le consensus est impossible pour un réseau complet de n processus **synchrones** dont f byzantins si $2 \leq n \leq 3f$.

La théorie dit que, bien sûr, si la majorité triche, on ne peut rien faire, mais que cela peut être problématique dès un tiers de "tricheurs" même sans perte de message.

Terminologie : Attaque Sybil

Dans un système ouvert comme Bitcoin, un attaquant peut créer autant d'identités qu'il le souhaite.

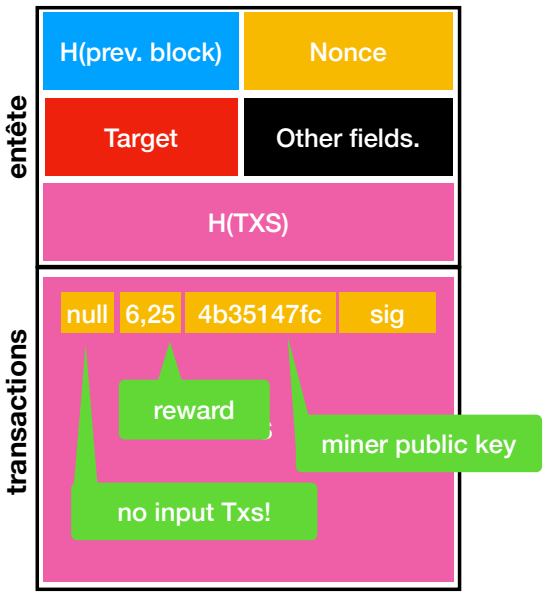
Attaque Sybil

Une *attaque Sybil* est une attaque par utilisation de multiples ("fausses") identités.

Du fait du résultat précédent, un mécanisme de régulation est indispensable.

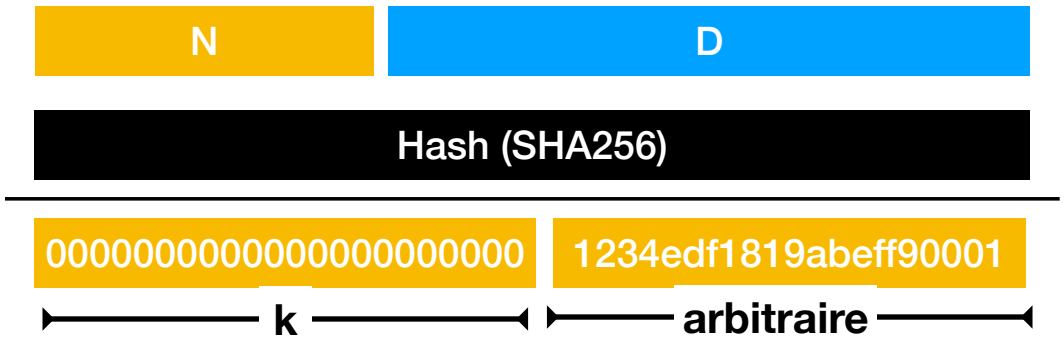
“Consensus” par preuve de travail

- Nouvelle loterie toutes les D minutes ($D = 10$ minutes)
- Le gagnant forme un nouveau bloc : ensemble des transactions validées
- Le gagnant reçoit une récompense
- En moyenne 1 vainqueur/loterie (mais il peut y avoir plusieurs vainqueurs ou pas de vainqueurs)



“Consensus” par preuve de travail

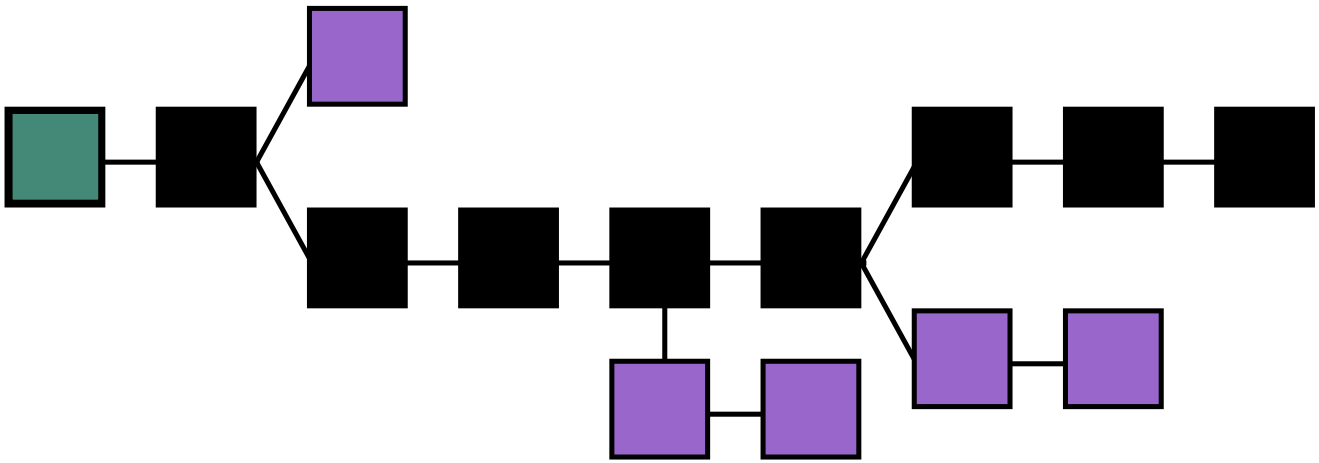
- Trouver un *Nonce* tel que le hash de l'entête du bloc est $\leq$ cible
- Tout le monde peut vérifier qu'un bloc est bien formé et gagnant
- Chance de gagner dépend de la puissance de hashage (#hashes/second)
- le gagnant est récompensé par les frais de transaction plus un montant fixe
- La difficulté de la cible est choisie de sorte qu'il y ait en moyenne un vainqueur en 10 minutes




```
N = 0;
tant que H(N|D) > 00..0F..FFF:
    N++;
retourner N
```

“Consensus” par preuve de travail

À un moment donné, il peut y avoir plusieurs gagnants et donc plusieurs branches.



Lorsqu’une branche devient plus longue qu’une autre, elle gagne et devient la seule branche valide.

Propriétés

- Le système deviendra normalement cohérent à moment donné
- Pas de délai garantissant qu’une transaction validée par un nœud sera définitivement inclus dans la blockchain.

Élection par preuve d'enjeu

Miner des blocs coûte très cher et consomme énormément d'énergie

$\Rightarrow$ utiliser un autre mécanisme de validation de bloc

Principe de la preuve d'enjeu

- Des nœuds mettent en gage une valeur en crypto
- Un des nœuds ayant misé est élu et décide du prochain bloc (et donc des transactions validées)

Propriétés souhaitables

- Au plus un seul vainqueur par ronde
- Le vainqueur doit pouvoir prouver qu'il a gagné
- On ne doit pas pouvoir prédire qui va gagner à l'avance

Section 5

Conclusion Provisoire

Centralisé vs Décentralisé

- Internet
- Cloud
- Chaîne de blocs / “crypto”
- Plateformes (GAFAM)

De la recherche fondamentale aux applications

Recherche fondamentale

- outils cryptographiques
- formalisation des propriétés souhaitables
- résultats d'impossibilité
- solutions algorithmiques

Applications performantes

- débit important
- latence faible
- haute disponibilité (tolérance aux pannes)

Technologies en évolution

Des technologies qui suivent les nouveaux besoins.

Des nouveaux types de SGBD

- NoSQL : SGBD compromettant la cohérence pour avoir les autres propriétés (Disponibilité, Tolérance au partitionnement et latence faible)
- newSQL : SGBD visant l'extensibilité tout en garantissant les propriétés ACID

Des nouvelles solutions distribuées

- CephFS : système de fichier distribué (libre)
- BigTable : SGBD haute performance (Google)
- Cassandra : SGBD NoSQL distribué (libre)
- ...

Un écosystème en mouvement

- Les “crypto” : NFT, Coin, Ethereum, ...
- De nouveaux algorithmes de Consensus byzantins
- Intégration de la protection de la vie privée
- Quel(s) modèle(s) économique(s) ?
- Quelle évolution de décentralisé vs centralisé ?
- ...

Pour aller plus loin

- Texte de Brewer 2012 : *CAP Twelve Years Later: How the “Rules” Have Changed*
- Faire un stage/TER dans l’équipe Distributed team DALGOrithms