IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Transactions distribuées

Fabien Dagnat

ILSD – FIAB – 4– 2025-2026

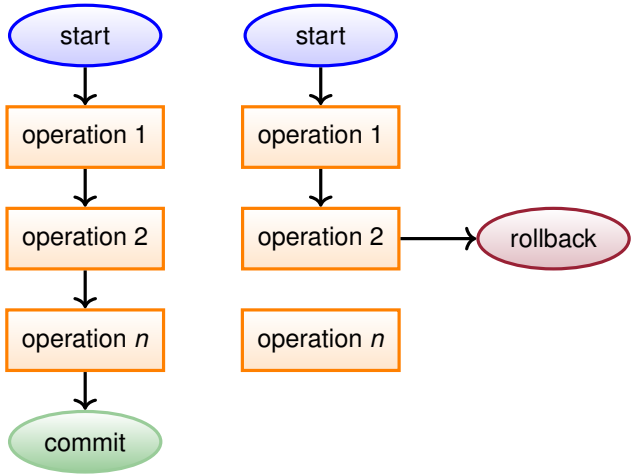
Qu’est-ce qu’une transaction ?

- Définition
  - Une transaction est un groupe d’opérations pour lequel la transaction est valide ssi
    - toutes les opérations s’exécutent correctement
    - aucune interférence ne se produit avec des opérations extérieure à la transaction
  - En général, les opérations manipulent des données (lecture ou écriture)
  - On passe d’un état cohérent à un autre état cohérent, les états intermédiaires ne peuvent pas être observé depuis l’extérieur de la transaction
- Exemples
  - Achat d’un trajet multiple en avion/train/bateau
  - Créer une réunion avec de multiples participants
  - Virements bancaires
  - Changer les droits d’un utilisateurs dans un SI

Propriétés ACID

- Atomicité : toutes les opérations sont exécutées ou aucune
  implique l'éventuelle annulation de certaines actions réalisées en cas d'échec
- Cohérence : si l'état avant transaction est cohérent celui après l'est également
  (qu'elle réussisse ou pas !)
- Isolation : le résultat est celui *comme si* la transaction était seule
  $T_1 \,||\, T_2 = T_1 ; T_2 = T_2 ; T_1$ (*serializability*)
- Durabilité : une transaction terminée ne peut être défaite que par une autre
  transaction

Schématiquement



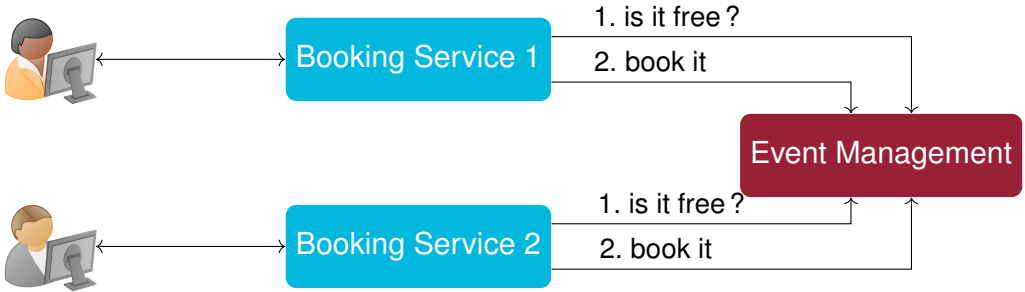
Pourquoi un Rollback / Abort ?

- Utilisateur change d'avis (cancel)
- Par programmation
  - en cas d'exception,
  - de détection d'incohérence,
  - de règle non respectée,
  - ...
- Pannes, protection système
  - *crash* système
  - *timeout*, interblocage

$\Rightarrow$ Il faut éventuellement défaire les modifications des opérations déjà exécutées

Un exemple

- Inspiré d'un article de blog
- Le principe de la réservation de ticket pour une place
  - on détermine si une place est libre
  - si elle est libre, on la réserve



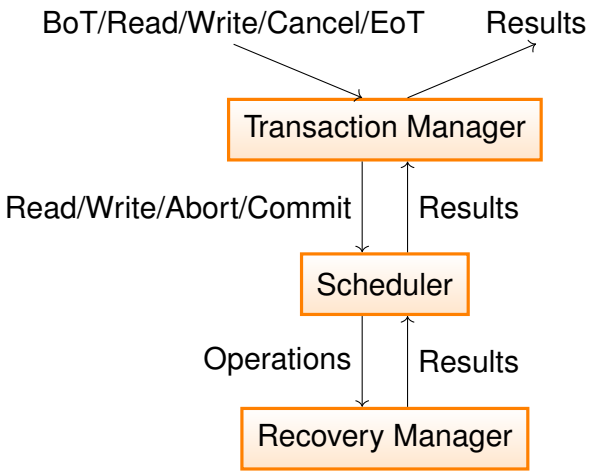
Usage et problèmes

- Usage
  - Exécution atomique et fiable même en cas de pannes
  - Exécution correcte en cas d'accès multiples
  - Gestion correcte des réplicas
- Problèmes
  - Consistence des données : contrôle sémantique / algorithmes de maintien de l'intégrité
  - Fiabilité : atomicité, durabilité, protocoles de commit global et de récupération locale
  - Contrôle des accès concurrents : isolation
  - Protocole de gestion des réplicas

Différents types de transactions

- Domaine d'application
  - Non-distribué vs. distribué
  - Transaction de compensation
- Temps
  - On-line vs batch
- Structure
  - Plate (ou simple) : ne contient que des opérations
  - Imbriquées : peut contenir des sous-transactions

Architecture

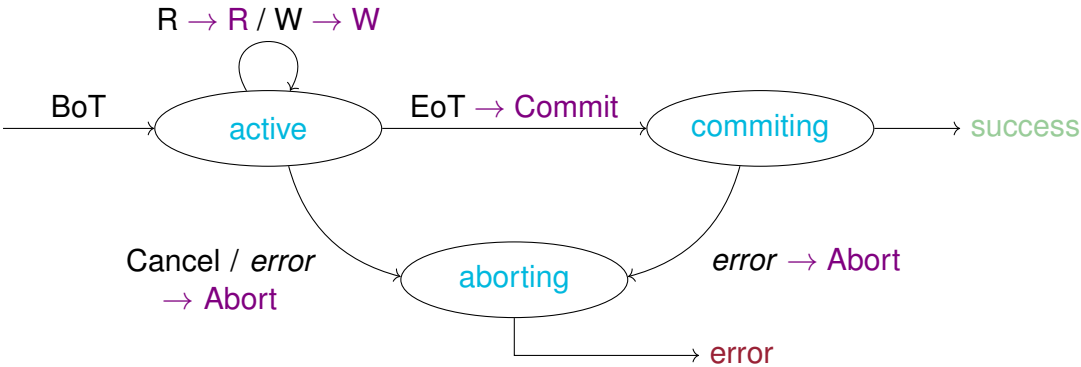


Gestion de l'exécution

Gestion de la concurrence

Gestion de la récupération

Cycle de vie d'une transaction



Recovery Manager

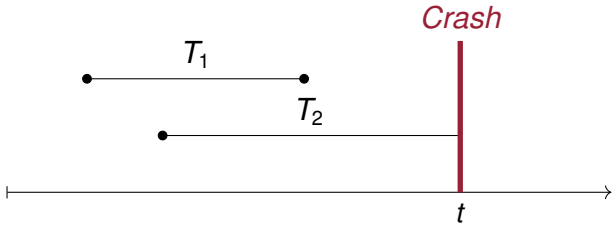
- Composant responsable de
  - la gestion de la récupération
  - des fonctions de *restart* et *rollback* en cas de *crash* (dans un état consistent !)
- Pour cela, il maintient une trace sous forme de journal, le log
  - c'est une collection d'entrées qui enregistre tous les changements dans leur ordre d'occurrence
  - composant uniquement additif, on ne peut rien retirer du log
  - il contient également des informations complémentaires (par exemple, les transactions)
- Le but du log est de permettre un retour à la cohérence en cas de *crash*

Exemple d'entrée d'un log

- Identificateur d'entrée
- Identificateur de transaction
- Type d'opération
- Éléments utilisées par la transaction pour l'opération
- Valeurs antérieures des éléments
- Valeurs postérieures aux opérations
- ...

Principe de la récupération

- si T_1 est indiquée comme validée, tous ses effets doivent être réalisés
  $\Rightarrow$ rejouer si nécessaire
- si T_2 est démarrée mais pas terminée, il faut l'annuler
  $\Rightarrow$ défaire si nécessaire



Procédure de récupération

- Cas d'un *rollback*
  - Remonte la chaine des entrées de la transaction dans le log
  - Pour chaque entrée, restaure l'ancienne valeur des données, et produit une entrée log de restauration
- Cas d'un *restart* après un *crash*
  - Rejoue la chaine des entrées du log
  - Pour chaque entrée, re-installe la nouvelle valeur des données, et produit une entrée Log pour enregistrer la restauration
  - Abandonne toutes les transactions non terminées du log
- Dans tous les cas, indique l'opération réalisée dans le log

Optimisation & Confiance

- Le log devient énorme !
- Il faut éventuellement « oublier » des éléments du log
  - attention aux contraintes légales !
- Pour optimiser, on peut aussi construire des points de reprise (*snapshot*)
  - permet de ne pas rejouer tout le Log
- Un *crash* peut se produire pendant un *restart* ou un *rollback*
  - ces fonctions doivent être idempotente
- Le log doit être sur un équipement différent (et répliqué sans doute)

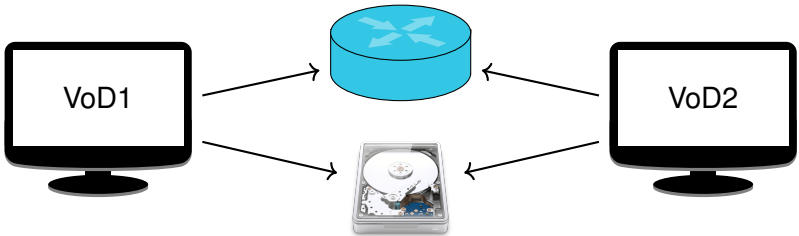
Scheduler

- Composant responsable de la gestion de la concurrence entre les transactions
- Parfois appelé *lock manager* car souvent utilise des verrous
- Rejète les requêtes de verrou en cas de conflit avec une autre transaction
- Un verrou n'empêche pas l'accès à une donnée, il empêche les conflits
- Mode de verrouillage
  - Verrou en écriture (W) ou en lecture (R)
  - Souvent, écriture exclusive, lecture partagée (attention néanmoins à l'isolation !)

Difficultés avec les verrous

- Ils doivent être bien utilisés
  - porte sur les bons éléments
  - sont bien demandés
- La granularité des verrous
  - de *gros* verrous entraînent moins de concurrence et sans doute des faux conflits
  - des verrous plus fins impliquent plus de concurrence mais plus de gestion de verrous
- Dans tous les cas, des risques d'interblocage

Accéder avec sûreté




```
VoD1 := while(true) { get(Rx); get(Dd); compute(); release(Rx); release(Dd); }
VoD2 := while(true) { get(Dd); get(Rx); compute(); release(Rx); release(Dd); }
```

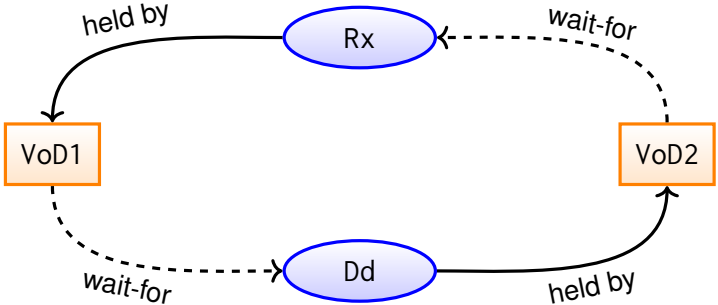
- Imaginez l'entrelacement

  ```
  VoD1.get(Rx); VoD2.get(Dd); VoD1.get(Dd); VoD2.get(Rx);
  ```

- Plus rien ne se passe, c'est un interblocage

Graphe de dépendance *wait-for*

- On peut construire dynamiquement un graphe de dépendance



- Un cycle est la marque d'un interblocage.

Solutions

- Prévention
  - par conception (par exemple ordonner la pose des verrous)
  - Avec preuves
- Détection, correction
  - Construire un graphe de dépendance
  - Rompre le cycle (par exemple abandon de la transaction la moins coûteuse)
  - attention, requiert un état global !
- *Timeouts*
  - plus simple
  - mais peut déclencher l'abandon sans raison

Un exemple d'algorithme : estampille

- À chaque transaction T_i est associée une estampille $ts(T_i)$
- Le *Transaction Manager* l'associe à chaque opération de T_i
- Chaque donnée x a une estampille d'écriture (*wts*) et de lecture (*rts*)
  - rts(x) = plus grande estampille de toutes les T_i ayant lu x
  - wts(x) = plus grande estampille de toutes les T_i ayant écrit x
- Les conflits sont résolus avec l'ordre des estampilles
  - $R_i(x)$ if $ts(T_i) < wts(x)$ then reject else {accept ; $rts(x) = ts(T_i)$}
  - $W_i(x)$ if $ts(T_i) < rts(x)$ or $ts(T_i) < wts(x)$ then reject else {accept ; $wts(x) = ts(T_i)$}

Qui donne l'estampille ? Qui connait les verrous ?

Transaction distribuée

- Ensemble de nœuds qui exécutent des transactions qui interfèrent
- Comment assurer l'atomicité ? l'isolation ?
- Il faut propager les effets et les échecs

$\Rightarrow$ Une transaction n'est pas une opération locale (en général)

- Il faut un coordinateur (centralisation) ou une coordination

Two-Phases Commit Protocol

1. Phase un : accord
   - Chaque participant vote *succès* ou *échec*
2. Phase deux : exécution
   - Si tous les votes sont *succès* alors *succès* de la transaction sinon *échec*

De nombreux algorithmes distribués fonctionnent par vagues.

Operations

- `canCommit?(T)`
  - Appel du coordinateur aux participants (vote)
- `doCommit(T)`
  - Appel du coordinateur aux participants pour autoriser le commit
- `doAbort(T)`
  - Appel du coordinateur aux participants pour abandonner
- `haveCommitted(T,p)`
  - Appel d'un participant au coordinateur pour confirmer le commit
- `getDecision(T)`
  - Appel d'un participant au coordinateur pour connaitre la décision si sans réponse depuis un certain délai (*crash* ou messages trop lents)

2PC Protocol : Phase 1

Phase 1 (vote)

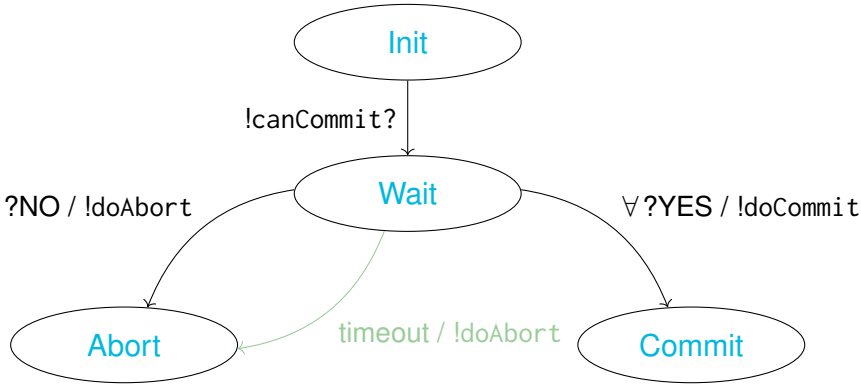
1. Coord. diffuse `canCommit?` à tous les participants
2. Chaque participant répond
   - YES : s'il peut valider et prépare le commit en sauvegardant le résultat
   - NO : s'il ne peut pas réaliser la transaction et abandonne immédiatement

2PC Protocol : Phase 2

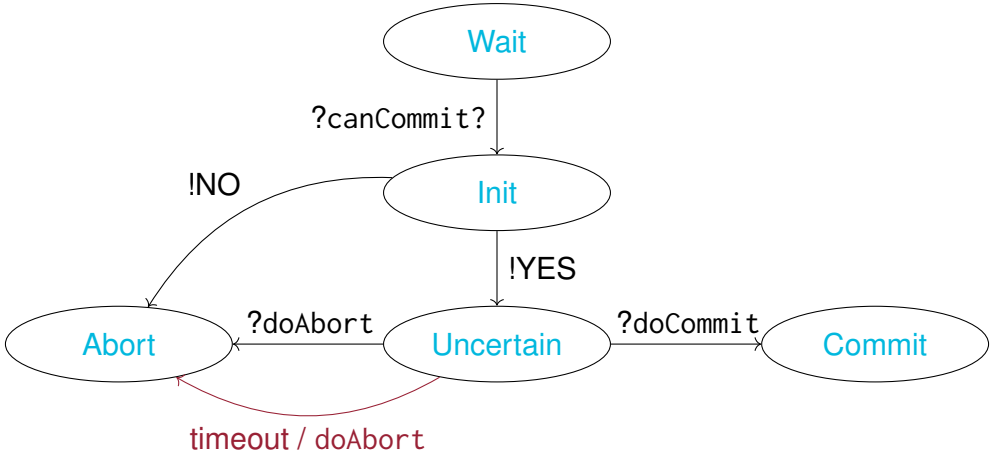
Phase 2 (finalisation)

1. Le coordinateur collecte les votes (y compris le sien)
   - Tous les votes sont YES, le coordinateur diffuse un `doCommit` à chaque participants
   - Il y a un NO, le coordinateur diffuse un `doAbort` aux votants YES
2. Les votants YES attendent la commande suivante, la réalisent, puis en cas de `doCommit` confirment avec un `haveCommitted`
   - pendant l'attente peuvent utiliser `getDecision`

Comportement du coordinateur



Comportement des participants



Pannes

- Que pensez-vous du *timeout* ? Peut-on abandonner après avoir voté YES ?
- On ne peut pas décider unilatéralement, le *timeout* interdit
- Si le processus apprend qu'un autre a fait un commit ou abort, il peut faire
  - le coordinateur a dû envoyer un message qui ne m'est pas arrivé
- Si le processus apprend qu'un autre est dans l'état *Init*, il peut avorter et votera NO à la nouvelle demande
  - le coordinateur a dû *crasher* pendant la diffusion du `canCommit`
- Si tous les processus sont dans l'état *Uncertain*, il faut attendre

Reprise après crash

- Coordinateur
  - Si la décision est enregistrée (log), confirmer aux participants (*push*) ou attendre un `getDecision` (*pull*)
  - Si la décision n'est pas consignée, répondre ou diffuser Abort
- Participants
  - Si crash après un YES, appeler `getDecision`
  - S'il faut commiter, récupérer l'état et effectuer le commit

Complexité du protocole 2PC

- Pour *n* participants, si tout ce passe bien
  - *n* `canCommit?`
  - *n* votes
  - *n* `doCommit`
- Si tout ne se passe pas bien (*crashes* de serveurs, pertes de message ...)
  - Les délais peuvent être très long et l'état *uncertain* durer longtemps
  - Un protocole en 3 vagues peut être mis en place

Conclusion

- On n'abandonne pas l'Atomicité et la Durabilité
- On peut relâcher l'Isolation et la Cohérence

$\Rightarrow$ Cohérence à terme