Réplication des bases de données

Stéphane Gançarski
Hubert Naacke

LIP6
Université Paris 6

Références

- Livre
  - Principles of Distributed Database Systems
- Articles
  - Eager Replication
    - [KA 2000]: Bettina Kemme, Gustavo Alonso: *A new approach to developing and implementing eager database replication protocols*. TODS 2000
    - www.cs.mcgill.ca/~kemme/publications.html
  - Lazy Replication
    - Esther Pacitti, Pascale Minet, Eric Simon: *Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases*. VLDB 99. Voir thèse de Cédric Coulon, LINA, Nantes
  - Articles Leganet
    - www-poleia.lip6.fr/~gancarsk/publications.html
- Cours
  - P.Valduriez

Plan

- Objectifs de la réplication
- Classification des solutions de réplication
  - Mono maître / Multi maître
  - Synchrone / asynchrone
    - Synchrone avec interaction linéaire/constante
    - Asynchrone optimiste/pessimiste
- Solutions existantes
- Solutions récentes pour cluster de BD
  - Réplication synchrone
  - Réplication asynchrone préventive
- Projet Leganet
  - Réplication asynchrone pour cluster de BD
  - Compromis performance/cohérence
  - Fraîcheur d'une réplique
  - Gestion des conflits
  - Equilibrage de charge

Définitions

- Réplique
  - copie d'un ensemble de données de référence.
- Site primaire (ou maître ou référence)
  - Application: accès en lecture/écriture,
  - Transaction de mise à jour: plusieurs ordres
    - select, update, insert, delete
- Site secondaire (ou esclave ou cible)
  - Application: accès en lecture seule
  - Requête: plusieurs ordres select
  - le SGBD (ou le réplicateur) gère la mise à jour
- Propagation: répercuter la mise à jour d'une donnée de référence, sur une donnée cible.

Accès à des données distantes

- Sans réplication
  - Toutes les applications accèdent au même SGBD
    - Surcharge du SGBD
    - Dégradation du temps de réponse
    - Transferts importants
    - Faible tolérance aux pannes
  - Simplicité
    - Transaction locale

Accès à des données répliquées

- Accès à plusieurs copies dans un environnement réparti
  - Parallélisme
    - Equilibrage de charge
    - Meilleur temps de réponse
    - Réduit les transferts de données
- Disponibilité d'une réplique
  - même lorsque la donnée de référence n'est pas disponible
  - Probabilité de panne plus faible
    - P(panne de N serveurs) = P (panne d'un serveur) ^{N}

Accès à des données répliquées: Inconvénients

- Gestion des mises à jour
  - Surcoût : échange de messages inter-sites
- Cohérence d'une donnée répliquée par rapport à la donnée de référence

Rappels:
Modèle de transaction
Contrôle de concurrence

Modèle de transaction

- Notations
  - Opérations de lecture (L), écriture (E), L ou E (Op)
    - Li(X) : lecture de X par la transaction Ti
    - Li,n(X) : n^{ième} lecture de X par Ti
    - Li,n(Xj): n^{ième} lecture de la réplique X sur le site j par Ti
  - Opérations de terminaison:
    - Ti valide, commit (ci)
    - Ti abandonne, rollback (ai)
  - Séquence d'opérations
    - Relation d'ordre (...,Opi, Opj,...) Opi $\prec$ Opj
    - Précédence si 2 trans différentes et au moins une écriture de la même donnée
      - Opi(X) $\prec$ Opj(Y) et i $\neq$ j et (Opi=E ou Opj=E) (Ti $\rightarrow$ Tj)
    - Séquence sérialisable $\Leftrightarrow$ graphe de précédence sans circuit
    - Graphe avec circuit $\Leftrightarrow$ incohérences ex: (Ti $\rightarrow$ Tj) et (Tj $\rightarrow$ Ti)

Types de conflits

- Conflits provoquant des incohérences
  - P0: Ecriture perdue
    - L1(X), E2(X), E1(X), c1
      - T1 écrase la mise à jour de T2.
  - P1: Lecture sale
    - E1(X), L2(X), (c1 ou a1)
      - T2 lit une donnée non validée
      - Abandon en cascade a1 $\Rightarrow$ a2
  - P2: Lecture non répétable
    - L1 (X), E2(X), c2, L1(X)
      - T1 lit 2 valeurs différentes de X
  - P3: Lecture biaisée (fantôme)
    - L1(X), E2(X), E2(Y), c2, L1(Y)
    - S'il existe une contrainte entre X et Y
    - alors T1 lit une version de X et Y qui ne satisfait pas la contrainte
  - P4: Ecriture biaisée (fantôme)
    - L1(X),L2(Y),E1(Y),E2(X)
    - S'il existe une contrainte entre X et Y
    - alors les 2 écritures ne respectent pas la contrainte

Niveaux d'isolation

- Standard ANSI SQL 92
  - Uncommited read
    - Évite P0
  - Commited read
    - Evite P0 à P1
  - Repeatable read
    - Evite P0 à P2
  - Serializable
    - Evite P0 à P4

Contrôle de concurrence: verrouillage

- Verrouillage
  - Verrou court (long) :relaché après opération (à la validation)
  - Ecriture: tjrs verrou exclusif (VX) long, pour éviter P0
  - Lecture: verrou partagé (VP). Dépend du niveau d'isolation
    - Uncommited read
      - Aucun verrou en lecture
    - Commited read
      - VP court
    - Repeatable read
      - VP long sur les tuples lus
      - VP court sur les tuples fantômes
    - Serializable
      - VP long

Contrôle de concurrence : mise en oeuvre

- Contrôle de concurrence strict
  - 2PL, attente pour les E
- Contrôle de concurrence relâché
  - réduit le nb de conflits L/E
  - Cursor stability [KA, 2000]
    - VP courts pour les lectures
    - VP long pour L avec intention d'écrire
      - cursor
      - select id, nom from Etudiant for update
    - + taux d'abandon réduit
    - n'évite pas les lectures non répétables
  - Snapshot Isolation [KA, 2000]
    - Lecture: aucun VP, maintenir plusieurs versions d'une donnée
      - Marquer une version avec le n° de trans qui a créé la version
      - Lire la version marquée de la dernière transaction ayant modifié X et validé.
    - Ecriture
      - Vérifier la version : si X modifié depuis le début de T, alors abandon
    - Implementé dans Oracle
  - Hybrid protocol
    - Txn de lecture seule : snapshot isolation
    - Txn avec mise à jour: 2PL
    - Sérializable

Propriétés de la réplication

- Cohérence forte garantie si toutes les répliques sont équivalentes à la référence
  - sérialisabilité quelque soit la réplique (1-copie serializability)
  - Exple: L1(A1),E2(A2),E1(A1),E2(A2)
    - sérialisable avec A1=A2=A ?
    - L1(A),E2(A),E1(A),E2(A) non sérialisable.
- Cohérence faible: divergence entre les répliques
  - Cohérence éventuelle (si arrêt des màj, alors convergence)
  - Niveau de fraîcheur d'une réplique
- Atomicité garantie en synchrone (ex. 2PC)

Pourquoi répliquer ? Motivations

- Améliorer les performances
- Améliorer la tolérance aux pannes
- Passage à l'échelle : augmenter le nb de répliques
  - Performance constante avec une charge croissante
  - Meilleure performance avec une charge constante
- Besoins des applications
  - ex: OLAP, BD mobile

Comment répliquer ? Moyens

- Caractéristiques des solutions de réplication
  - Selon l'endroit où les applications envoient leurs transactions
    - un seul maître reçoit toutes les transactions
    - plusieurs maîtres: une txn est envoyée sur un des maîtres.
  - Selon le moment où les mises à jour sont propagées (quand répliquer ?)
    - propagation synchrone ou asynchrone des mises à jour
  - Transactions réparties ou non (si réplication totale)

Réplication mono-maître

- un seul maître par donnée:
  - référence (R ou S)
- plusieurs esclaves : r_i, s_i
  - (R) (r1) (r2)
  - (R) (S) (r1,s1)
  - (R), (S), (r1, s1), (r2, s2)
- un site esclave peut être maître pour une autre donnée
  - (R) (r1, S) (s1)

Réplication multi-maîtres

- Plusieurs maîtres pour une donnée (R1, R2)
- Mises à jour sur R1 et R2
- Pb: propagation des mises à jour. Plus difficile à sérialiser que mono-maitre

Propagation des mises à jour

- Les mises à jour de R sont réparties sur plusieurs maîtres (R1) … (Rn)
- Une réplique (r) doit recevoir toutes les mises à jour reçues par les maîtres
- Propagation
  - gérée par le SGBD réparti
  - Synchronisée, ou non, avec la transaction

Propagation synchrone

- Avant la validation de la transaction
- L'application obtient une réponse **après** la propagation
  - 1)transaction locale → 2) propagation → 3) validation → 4) réponse
- Propagation : nb de messages échangés entre sites
  - Immédiate après chaque opération (L,E)
    - 1 message par opération (L/E): interaction linéaire
  - Différée juste avant la fin de la transaction
    - 1 message par transaction : interaction constante
- Contenu du message: SQL ou Log si disponible
- Validation: décision prise
  - par plusieurs sites (vote, ex 2PC)
  - par chaque site séparément (sans vote)

Propagation asynchrone

- L'application obtient une réponse **avant** la propagation
  - 1) transaction locale → 2) validation locale → 3) réponse
  - 4) propagation → 5) validation
- Pessimiste :
  - Mises à jour et propagations faites dans un ordre sérialisable prédéfini pour les transactions
- Optimiste :
  - Ordre entre transactions calculé en cours d'exécution en fonction des opérations commutatives, conflictuelles. Possibilité d'abandon

Classification des solutions

- Monomaître / Multimaître
- Asynchrone / Synchrone
  - Synchrone : avec intéraction constante / linéaire
  - Asynchrone : optimiste / pessimiste
- Validation: avec vote / sans vote
- Termes
  - Primary copy = Monomaître
  - Update Everywhere = Multi-maître
  - Eager Replication = Repl. avec propagation synchrone
  - Lazy Replication = Repl. avec propagation asynchrone

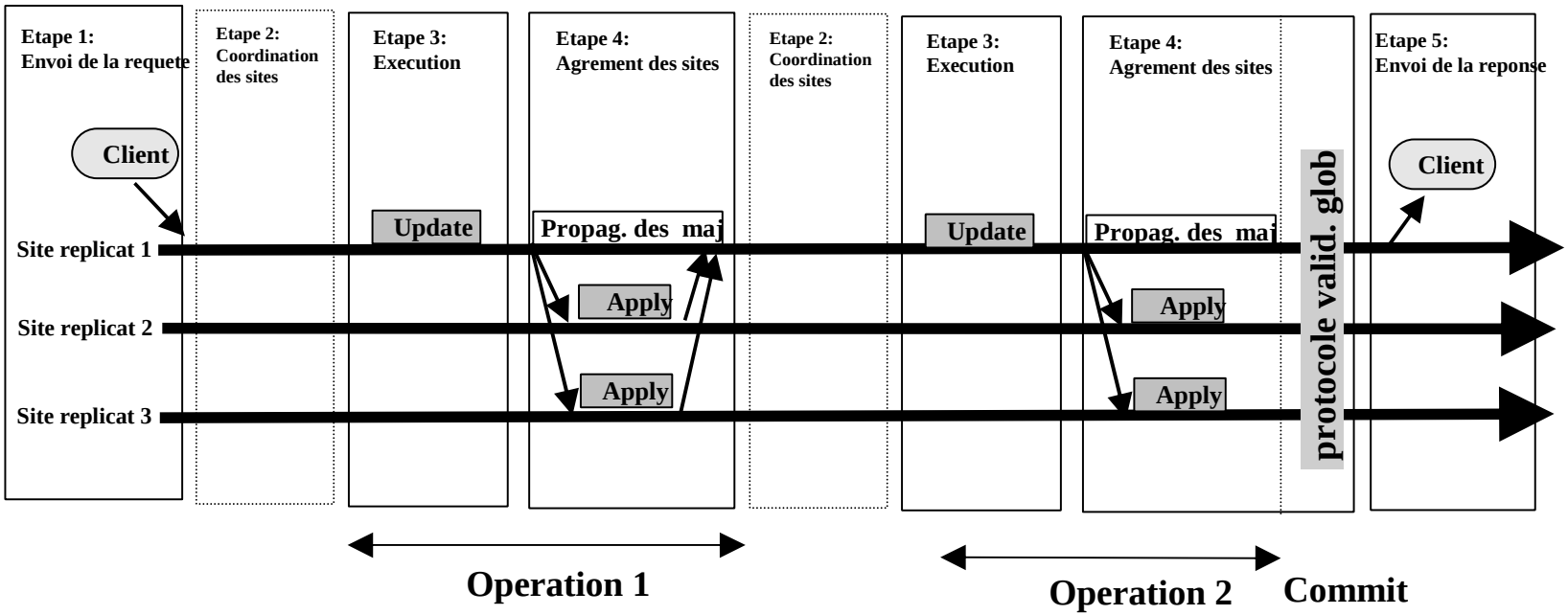
Réplication avec propagation synchrone

R1: Réplication mono-maître et Propagation synchrone

- Lorsque le maître reçoit une demande de :
  - Lecture: traitement local, réponse
  - Ecriture: traitement local, transmet la demande d'écriture aux cibles dans l'ordre FIFO, réponse immédiate
- Lorsque une cible reçoit une demande de :
  - Lecture: traitement local, réponse
  - Ecriture venant d'une application : refus
  - Ecriture venant du maître : traitement (dans l'ordre FIFO)
- Interblocage
  - Détection locale sur le maître
- Validation avec vote : 2PC
  - une réplique peut remplacer le maître défaillant : «hot standy»
- Validation sans vote : réplique = sauvegarde à froid
  - S'assurer que chaque transaction est propagée

R1: protocole

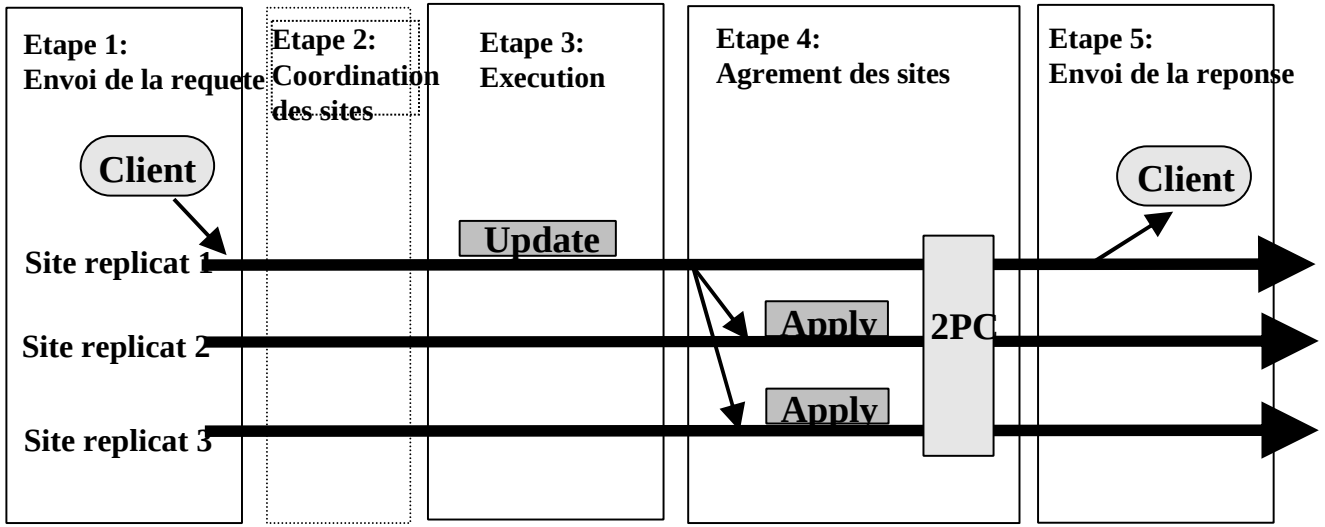
a) interaction linéaire



R1 : protocole

b) interaction constante

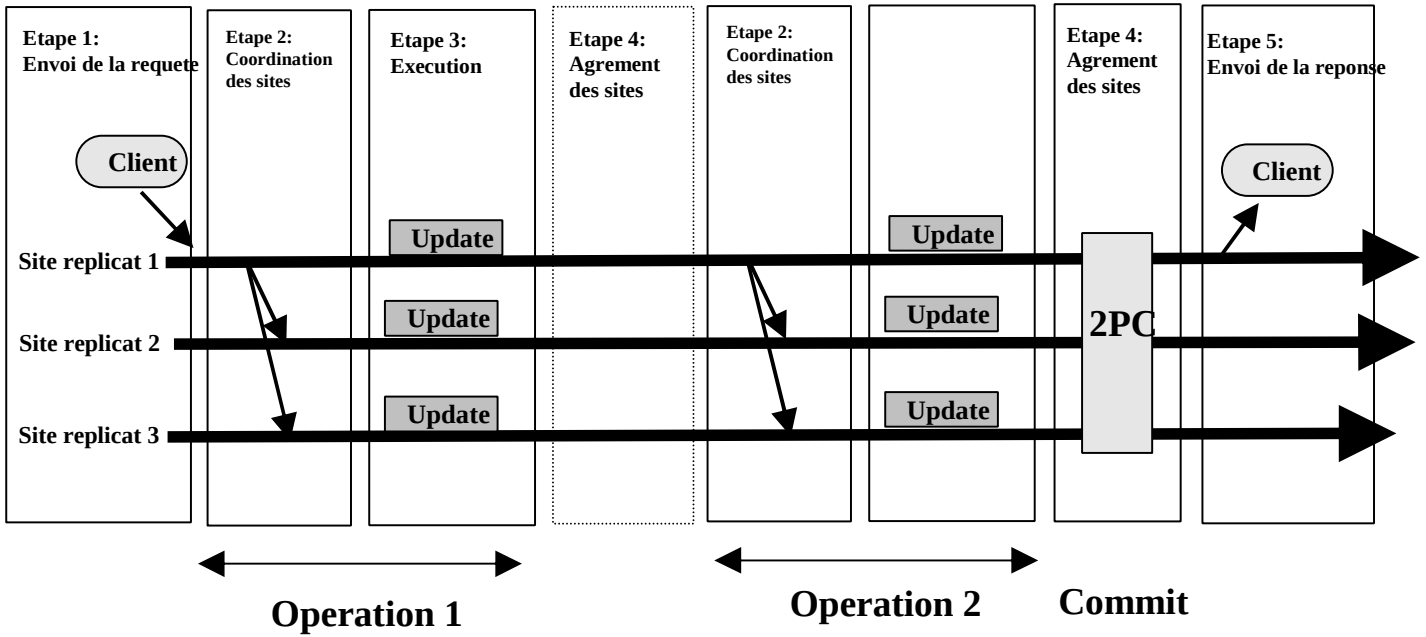
- Un message par transaction
- Màj sur shadow copy



R2a : Multi-maîtres, Synchrone, interaction linéaire

- Lorsque un maître reçoit une demande de :
  - Lecture: verrou P local, lecture locale, réponse
  - Ecriture venant d'une application
    - verrou X local, écriture locale, transmet la demande d'écriture aux autres maîtres dans l'ordre FIFO, attente des OK
  - Ecriture venant d'un autre maître
    - Verrou X local, écriture locale, réponse OK au maître
  - OK venant de tous les maîtres: réponse OK à l'application
- Interblocages
  - Détection locale insuffisante
  - Nb d'interblocages augmente avec le nb de sites
- Validation avec vote
  - Ex 2PC: protocole coûteux
- Validation sans vote
  - un total order broadcast pour chaque opération (L,E)
  - Coût élévé en nb de messages échangés

R2a : protocole

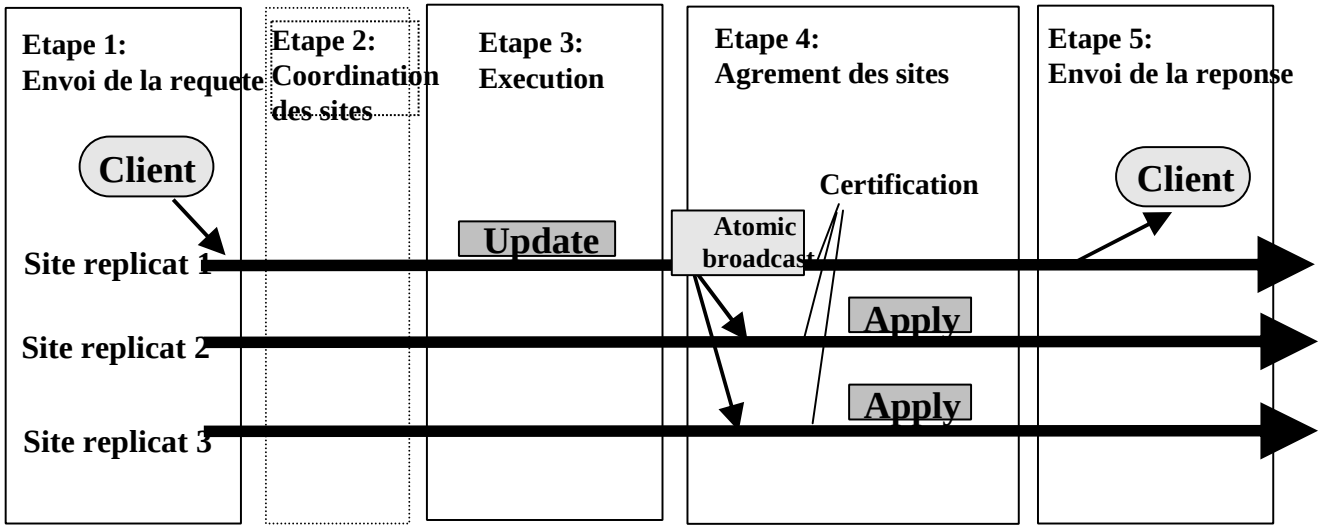


R2b: Multi-maîtres, Synchrone, interaction constante

- Validation sans vote
  - Traitement déterministe
    - Commencer par ordonner la transaction
      - total order broadcast
  - Traitement non déterministe
    - Finir par une phase de certification déterministe
      - Abandon si conflits détecté
- Validation avec vote

R2b: Protocole

- broadcast: les opérations arrivent dans le même ordre sur **tous** les sites
- Certification : le site certifie qu'il peut exécuter les opérations dans cet ordre



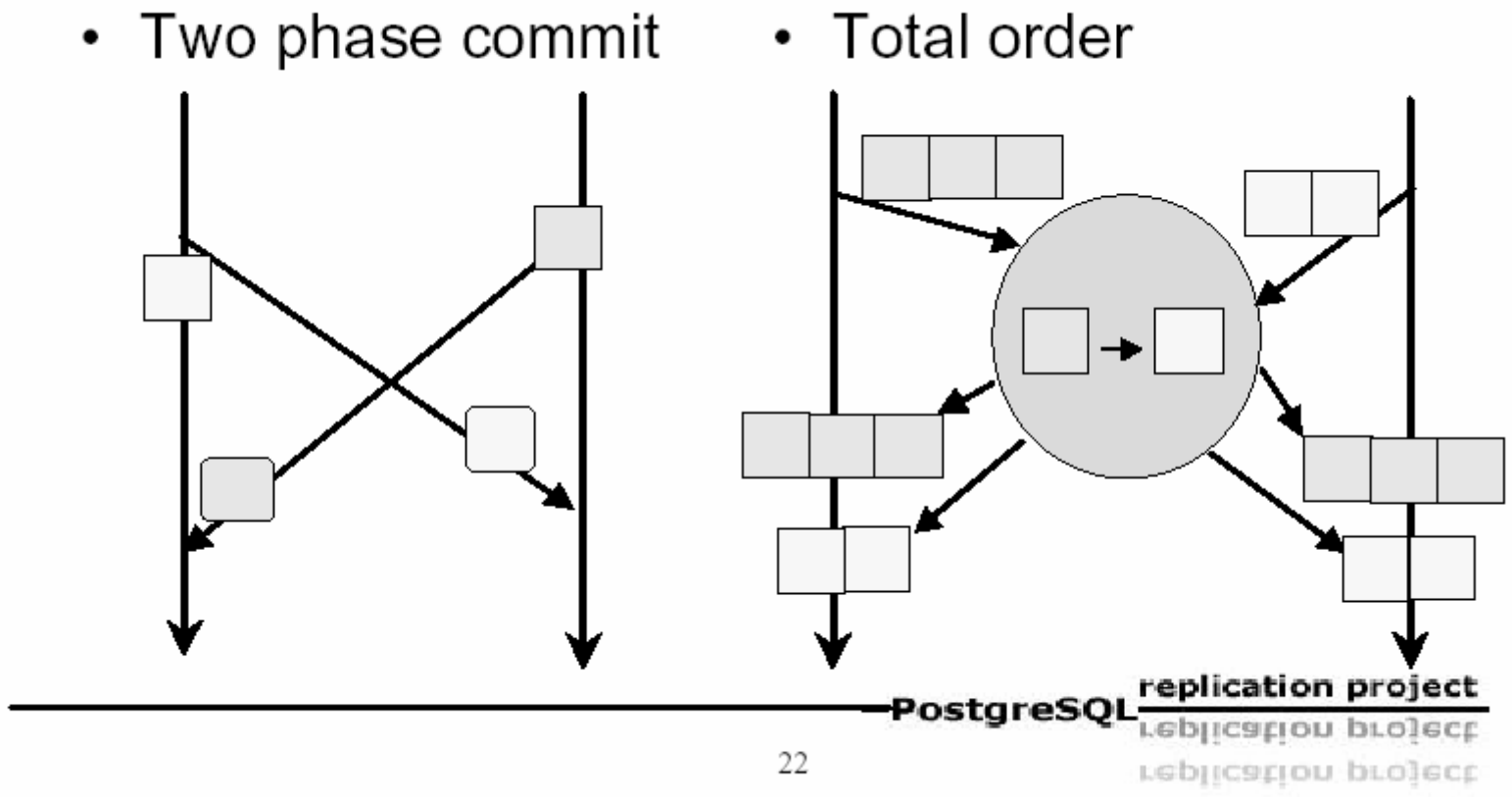
Replic Synchrone: Mise en œuvre

- multicast en FIFO sur les cibles
- Interne au SGBD: selon la granularité
  - pour chaque ordre DML : Trigger
  - pour chaque procédure stockée: Adaptateur de procédure
    - Avantage: moins de messages qu'avec des triggers
    - Inconvénient: traitement en série des procédures pour garantir la sérialisabilité
- Externe au SGBD: réplicateur
  - Pour chaque ordre DML ou chaque transaction

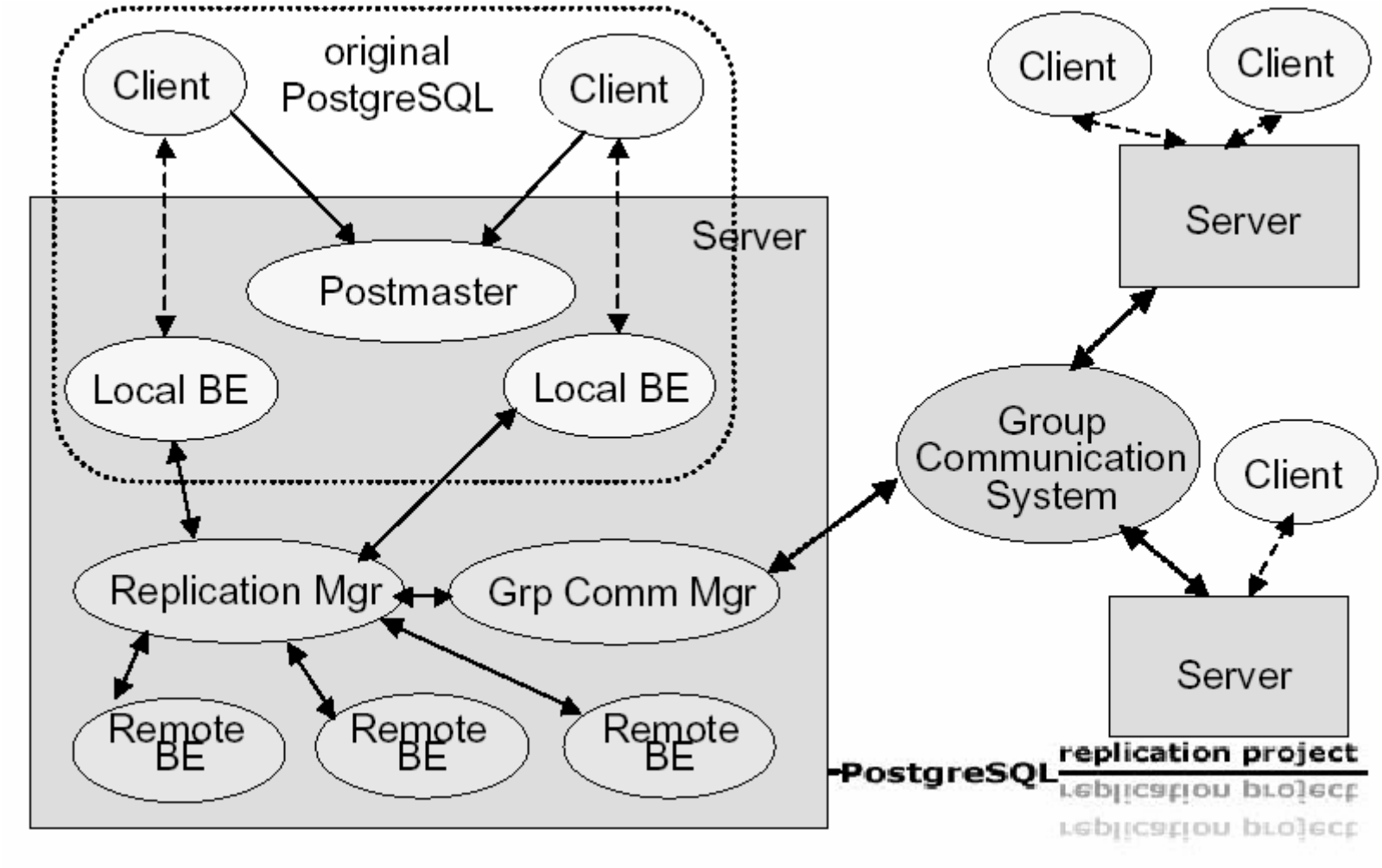
Postgres-R

- Réplication synchrone, multi-maîtres, interaction constante
- Total order broadcast
  - Après le traitement local des opérations
- interaction constante
  - 2 messages par txn (propager les écritures + confirmation)
- Mise à jour des répliques: 2 solutions
  - Propager les tuples modifiés
    - message volumineux, mise à jour rapide
  - Propager les ordres SQL (insert, update, delete)
    - Message compact, mise à jour lente
- Avantages
  - Pas de 2PC : remplacé par le total order broadcast
  - Validation possible avant la fin de la propagation

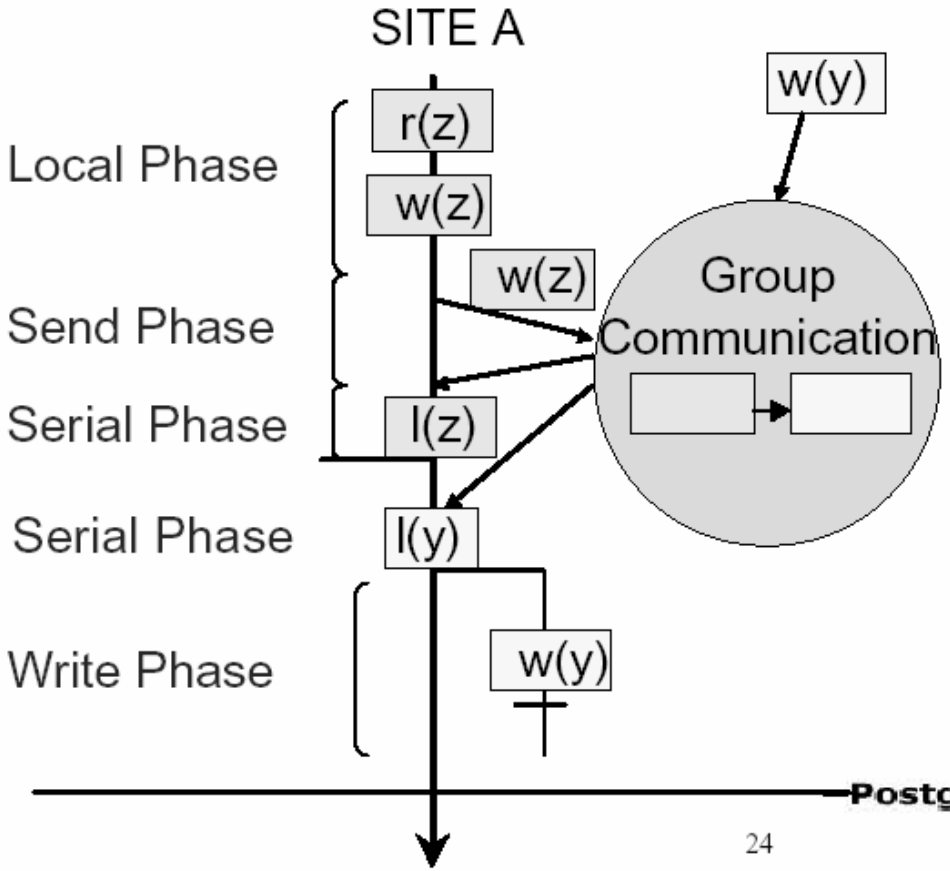
Postgres R
total order broadcast



Postgres R: Architecture

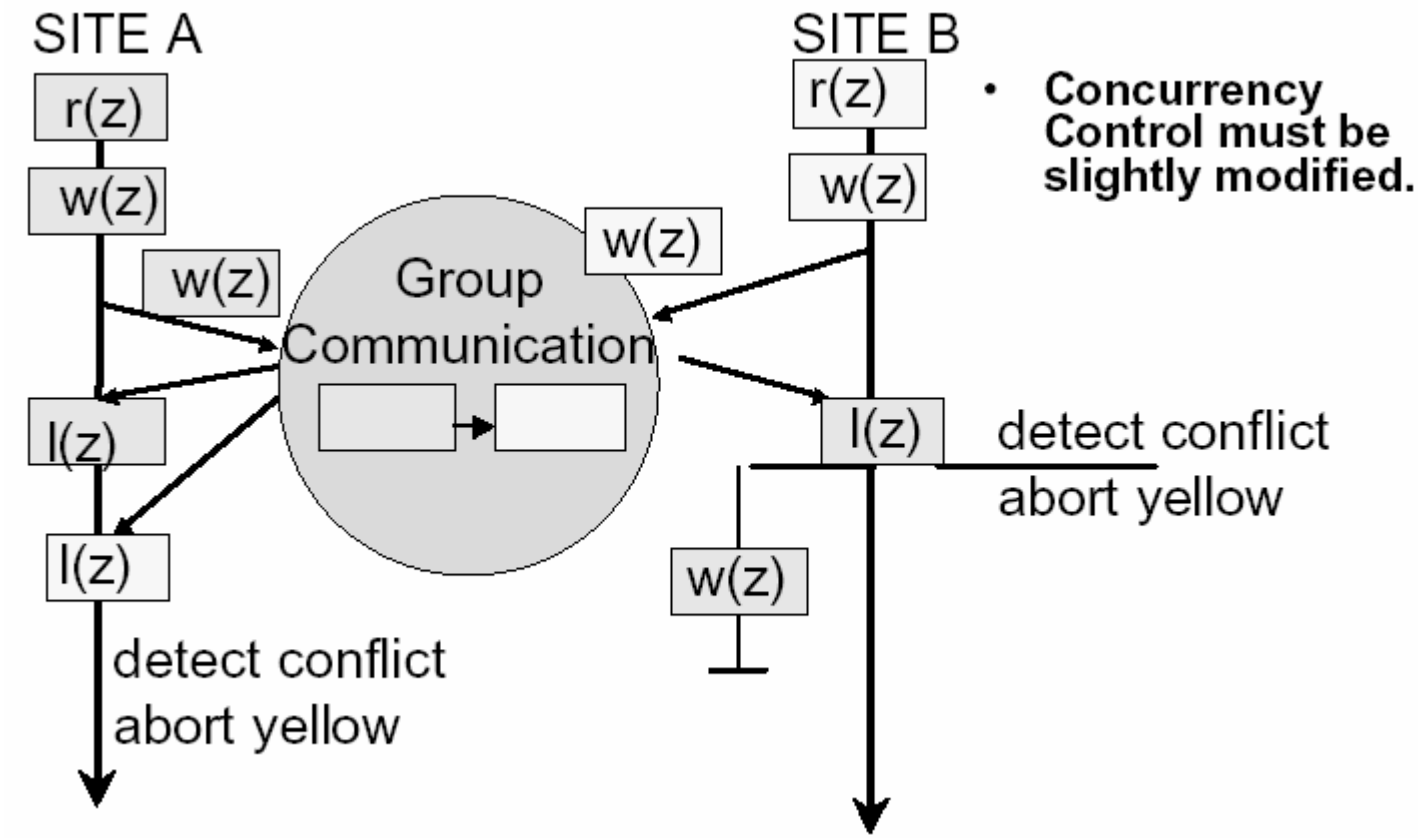


Postgres R: Basic Protocol



- All operations are first performed locally at a single site.
- Writes are sent in one message to all sites at the end of transaction.
- Write messages are totally ordered.
- Serialization order obeys total order (same serialization order at all sites).
- When receiving remote writes, they must be applied

Postgres R: Conflicts



Postgres-R: protocole

- Etape 1: traiter toutes les opérations de Ti
  - Lecture : locale
  - Ecriture Ei
- Etape 2: propager les écritures Ei
  - Message diffusé avec ordre total
    - Envoyé à tous les sites, site expéditeur inclus
    - Détermine l'ordre des transactions
- Etape 3: Dès réception de Ei : verrouillage
  - Tester les conflits entre Ei et L/E locales
    - Si conflit : éviter les interblocages et défaut de sérialisation
      - si txn locale en cours (étape 1 ou 2), abandonner txn locale. Si étape 2, diffuser msg d'abandon
      - Sinon abandonner Ti et diffuser la décision (sans ordre)
    - Si pas de conflit : verrou accordé ou mise en attente.
  - Message de confirmation : Validation si locale/ abandon si conflit
    - Diffusion sans ordre
- Etape 4: Appliquer les écritures
- Etape 5: terminaison
  - Txn locale valider et relacher les verrous
  - Txn distante: valider dès réception du message de confirmation

Exemple: étapes 1 et 2

- Configuration
  - 2 nœuds maîtres: N1, N2
  - Données (X,Y) sur N1 et N2
- Exécution T1 sur N1, T2 sur N2
  - L1(X), L2(Y), E1(Y), E2(X)
  - Non sérialisable
  - L1(X) sur N1: VP(X,T1)
  - L2(Y) sur N2: VP(Y,T2)
  - E1(Y) sur N1
    - Exécuter E1(Y)
    - Diffuser WS1 = E1(Y)
  - E2(X) sur N2:
    - Exécuter E2(X)
    - diffuser WS2 = E2(X)

WS : Write Set

Exemple : Etapes 3 à 5

- Réception de WS1 sur N1
  - ordre T1 puis T2
  - VP(X,T1c), VX(Y,T1c)
  - Appliquer WS1, suppr les verrous de T1
  - Diffuser c1
- Réception de WS1 sur N2
  - Ordre T1 puis T2
  - Remplacer VP(Y,T2) par VX(Y,T1c)
  - Abandon: diffuser a2
- Réception de WS2 sur N1
  - Attente_VX(X,T2)
- Réception de a2 sur N1: supprimer le VX en attente
- Réception de c1 sur N2
  - Appliquer WS1, suppr les verrous de T1

Postgres-R : protocoles non sérializables

- Protocole modifié pour réduire le taux d'abandon
  - Cursor stability
    - Etape 1: verrous VP court pour les lectures seules
    - Etape 3: mettre en attente la demande VX(X)
      - si tous les VP sur X sont courts
      - ou s'ils sont long et appartiennent à des txn prêtes à valider
  - Snapshot isolation
    - Etape 1: reconstruire une version isolée
    - Etape 2: diffuser les écritures + estampille début txn
    - Etape 3:
      - abandonner Ti si
        - Ti demande un verrou sur X
        - et X modifié par Tj plus récente (estampille Tj > estampille Ti)

Réplication synchrone: autres protocoles

- Problème lié au verrouillage
  - ROWA: Read Once Write All
    - Lecture: 1 verrou (local), Ecriture: verroux sur les N sites
  - transmettre une écriture à tous les sites puis attendre
  - Bloquant dès qu'un site est en panne.
- Tolérer la panne d'un (ou plusieurs) sites parmi N
  - transmettre seulement aux sites disponibles
  - ROWAA: ROW All Available
- Tolérer la panne de communication entre 2 sites
  - Quorums

ROWAA

- Modèle
  - Un site en panne peut être restauré
  - Pas de panne de communication
- U: ensemble des sites disponibles
- Algo:
  - Première écriture transmise à tous les sites, attendre un délai fixé (t)
  - U = ens. des sites ayant répondu avant la fin du délai t
  - Ecritures suivantes transmises à U
  - Validation: seuls les sites de U participent au 2PC
    - Vérification : si U a changé alors abandonner
- Cas pris en considération
  - site disponible n'appartenant pas a U
    - Lorsque la réponse à la première écriture > t
  - site restauré après le début de la transaction
    - Ne doit pas manquer la transaction en cours

Quorum

- Modèle
  - Tolère une panne de communication
  - Tolère les pannes modifiant la topologie du réseau
    - Si tous les sites détectent instantanément le changement de topologie
- Définition
  - Un quorum est un ensemble de site. L'intersection entre 2 quorums quelconques est non vide.
- Majorité simple: quorum de site
- Majorité pondérée: consensus
- Structure logique
  - Ex: Matrice de sites : un quorum formé d'une ligne et d'une colonne.

Quorum pondéré

- Protocole
  - Chaque réplique a un poids
  - N = poids total des répliques
  - Soient PL (resp. PE) le poids nécessaire pour traiter une lecture (resp. une écriture), tels que
    - 2PE > N évite les conflits E/E
    - PL+PE > N évite les conflits L/E
  - Un quorum en lecture (reps. Écriture) est un ensemble de répliques tel que :
    - somme des poids >= PE (resp. PL)
  - Généralise le ROWA: PL=1 et PE=N

Réplication avec propagation asynchrone

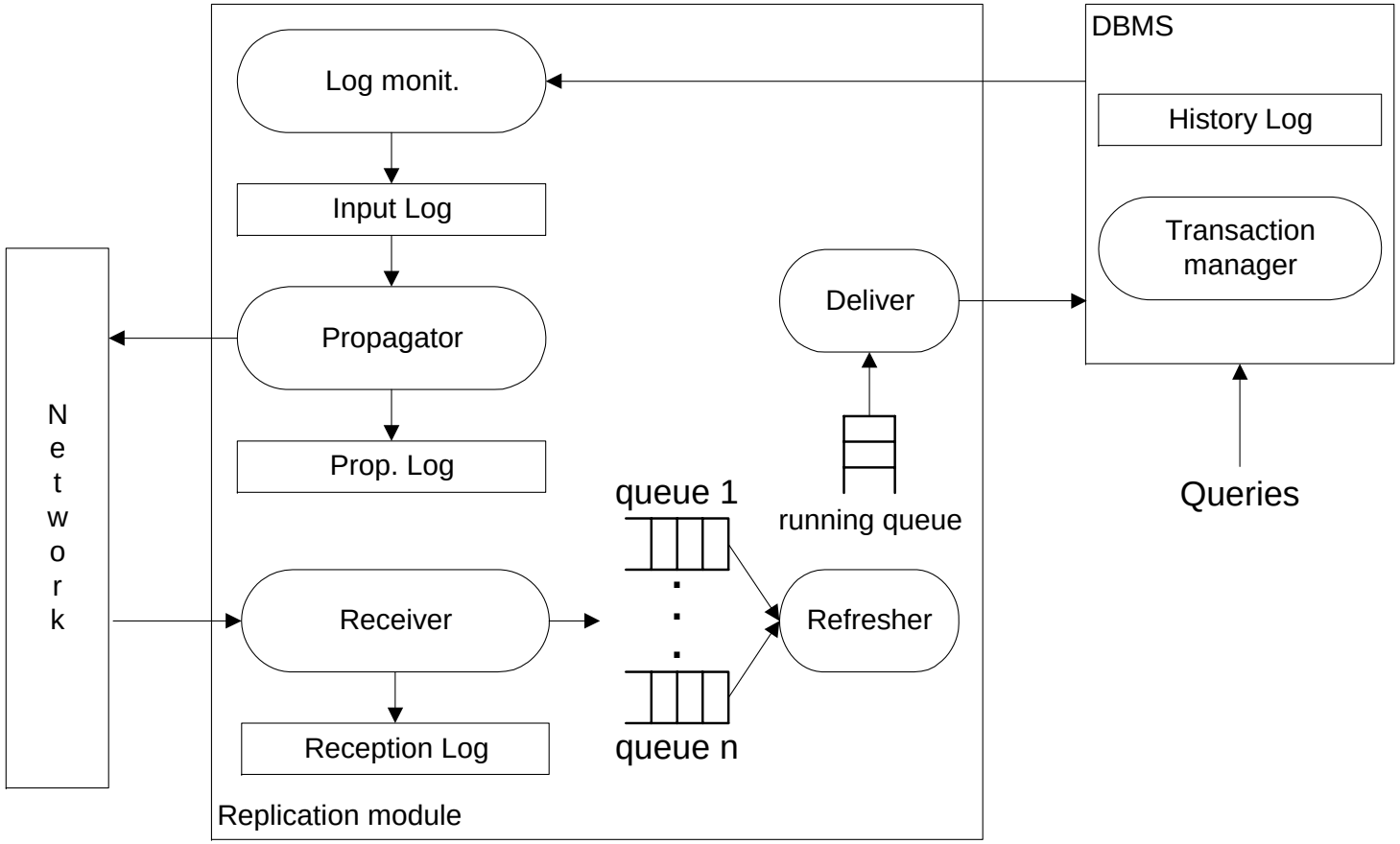
R3 : Réplication asynchrone mono-maître

- Lorsque un maître reçoit une demande de :
  - Lecture : lecture locale, réponse
  - Ecriture : écriture locale, réponse
  - Validation: locale
- Au déclenchement de la propagation
  - transmettre les écritures aux cibles
- Lorsque une cible reçoit une demande de
  - Lecture: locale, réponse
  - Ecriture venant d'une application : refus
  - Ecriture venant du maître: traitement (dans l'ordre FIFO)
  - Validation d'une txn en lecture seule: validation locale
- Interblocages
  - Détection locale suffisante

R4a: Réplication asynchrone multi-maîtres solution pessimiste (= préventive)

- Objectif: garantir la sérialisabilité
- Dépend de la configuration des
  - Graphe de configuration
    - Nœud = site
    - Arc non-orienté = maître-cible
- Graphe acyclique: monomaître
  - sérialisabilité garantie
- Cycle : multimaître
  - Mise en série des txn: un ordre global commun à tous les sites
    - Nécessite un temps de comm. intersite borné (tmax)
    - Exécuter les transactions dans l'ordre global
  - Classe de conflit : ensemble de txn conflictuelles
    - Traiter en parallèle 2 txn dans des classes distinctes

Réplic. Async. Préventive : Mise en oeuvre



Réplic. Async. Préventive : Solution

- Solution proposée
  - On reporte la mise à jour sur les copies secondaires: C + Max + ε
    - C est l'estampille globale (heure d'exécution de la transaction sur la copie primaire)
    - Max est le temps maximal d'arrivée d'un message d'un nœud à un autre
    - ε correspond au temps de traitement d'un message
  - Chaque nœud esclave possède une file pour chacun de ses maîtres

Réplic. Async. Préventive : Exemple

- Un nœud avec 2 files q(i) et q(j) vides. Max=10 et ε=1
- t0=110
  - Arrivée sur q(i) d'un message avec comme valeur C=105
  - q(i) = {105}, q(j) = {}
  - On élit q(i) avec un timeout à 116 = (105 +10+1)
- t1=112
  - Arrivée sur q(j) d'un message avec comme valeur C=103
  - q(i) = {105}, q(j) = {103}
  - On élit q(j) avec un timeout à 114 = (103+10+1)
- t2=114
  - Expiration du timeout, le message est retiré de q(j) et est placé dans la file d'exécution
- Amélioration
  - Réduire le timeout lorsqu'une file contient au moins un message de chaque nœud.
  - Approche plus optimiste avec possiblité d'abandon (Thèse de C. Coulon)

R4b : Réplication asynchrone multi-maîtres
Solution optimiste

- Les demandes de lecture, écriture, validation sont traitées localement
- Au déclenchement de la propagation: transmettre les écritures aux autres maîtres.
- A la réception d'une demande d'écriture venant d'un autre maître
  - Détecter les conflits et les résoudre
  - Traiter l'écriture localement.

Déclenchement de la propagation asynchrone

- Périodique
- A l'initiative
  - du maître : push
    - Complet: déclenchement sur toutes les répliques
    - Partiel: déclenchement pour certaines répliques
  - d'une cible: pull
- Opérations propagées
  - Complète : toutes les mises à jour sont propagées
  - Partielle: seules les mises à jour de certaines txn sont propagées.

Réplication asynchrone: tolérance aux pannes

- Validation locale: pas de problème en cas de panne d'un autre site
- Scénario d'exécution
  - Site A
    - exécute et valide T
    - propage les mises à jour au site B
  - Si panne de B
    - A conserve les mises à jour et retente périodiquement la propagation
    - Après restauration de B : succès
    - Garantir que B a reçu exactement une fois les mises à jour
      - Propagation dans une file persistante
  - Si panne de A après validation et avant propagation
    - Propagation après restauration de A.

Gestion des conflits

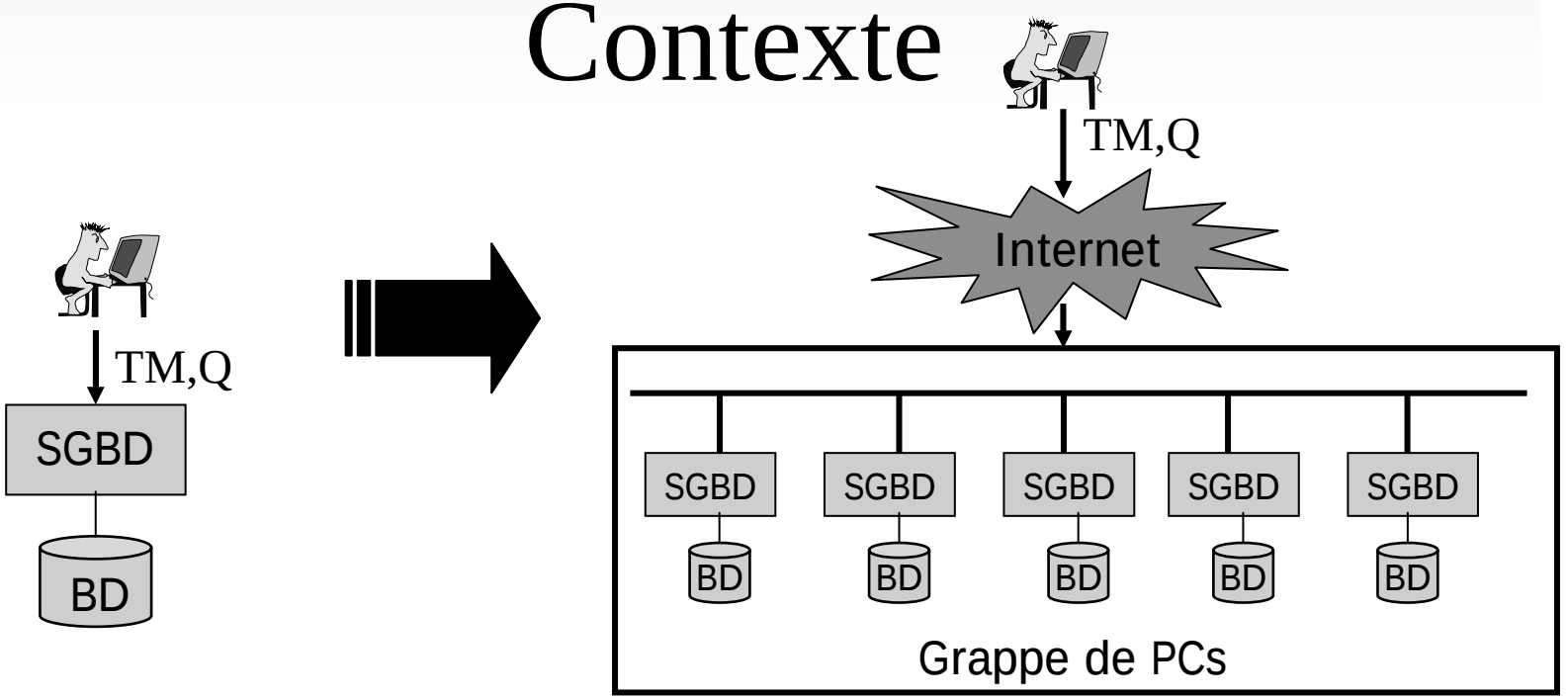
- Détection
  - Conflit de mise à jour:
    - Une réplique subit une modification avant de recevoir une autre modification en cours de propagation
    - Ancienne valeur (site orig) != valeur courante (site récepteur).
  - Conflit d'unicité
    - La propagation d'une mise à jour enfreint une contrainte d'intégrité (clé primaire ou unicité)
    - la même valeur d'un attribut unique est utilisée par 2 transactions.
      - 2 insertions en conflit
      - 2 update en conflit
      - conflit update/insert
  - Conflit de suppression:
    - Une transaction tente de modifier ou supprimer une ligne qui a été déjà supprimée sur un autre site
- puis Résolution

Résolution des conflits

- Résolution avec intervention humaine
  - Notifier l'admin. de la BD
- Résolution automatique
  - Écraser, ignorer
  - Priorité de site
  - Estampille: garantie la convergence
    - plus récente
    - plus ancienne
  - Pour les types numériques:
    - Exple: ancienne valeur x=2
    - nouvelles valeurs x=4, x=6
    - Moyenne x=5
    - Valeur min ou max
    - Additive: x=2+ (4-2)+ (6-2) = 8 (pour application débit/crédit)
- Méthode spécifique
  - Procédure stockée

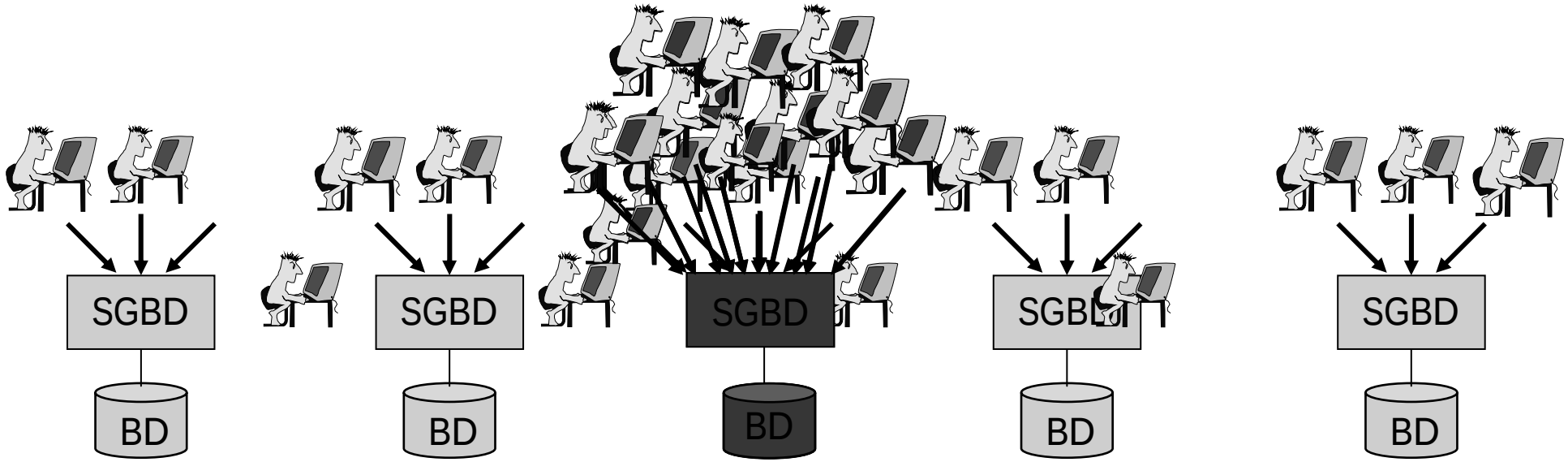
Contrôle de Qualité des Données Répliquées dans une Grappe de PCs

Contexte



- L'application envoie des transactions
  - Transactions de mise à jour (TM), de courte durée.
  - Requêtes en lecture seule (Q), de longue durée.
- Exécuter les transactions sur la grappe a pour avantages :
  - Faible coût, flexibilité, disponibilité, simplicité, ***efficacité***.

Problème 1 : performances

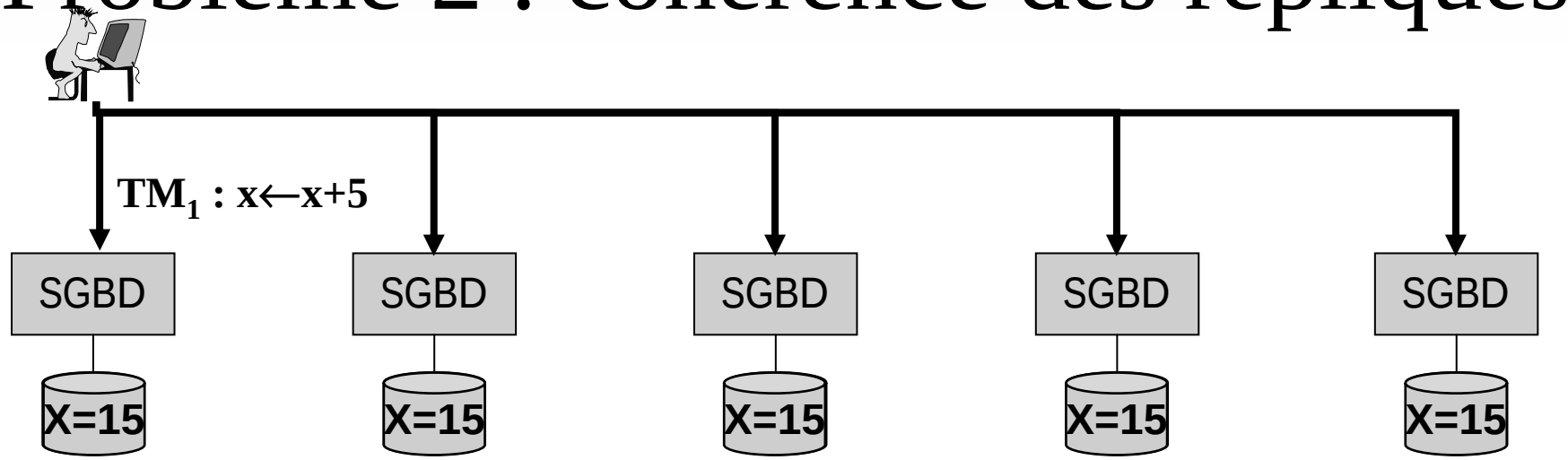


- Réplication des données
- Equilibrage de charge

⇒ *Répartition des transactions sur les nœuds de la grappe*

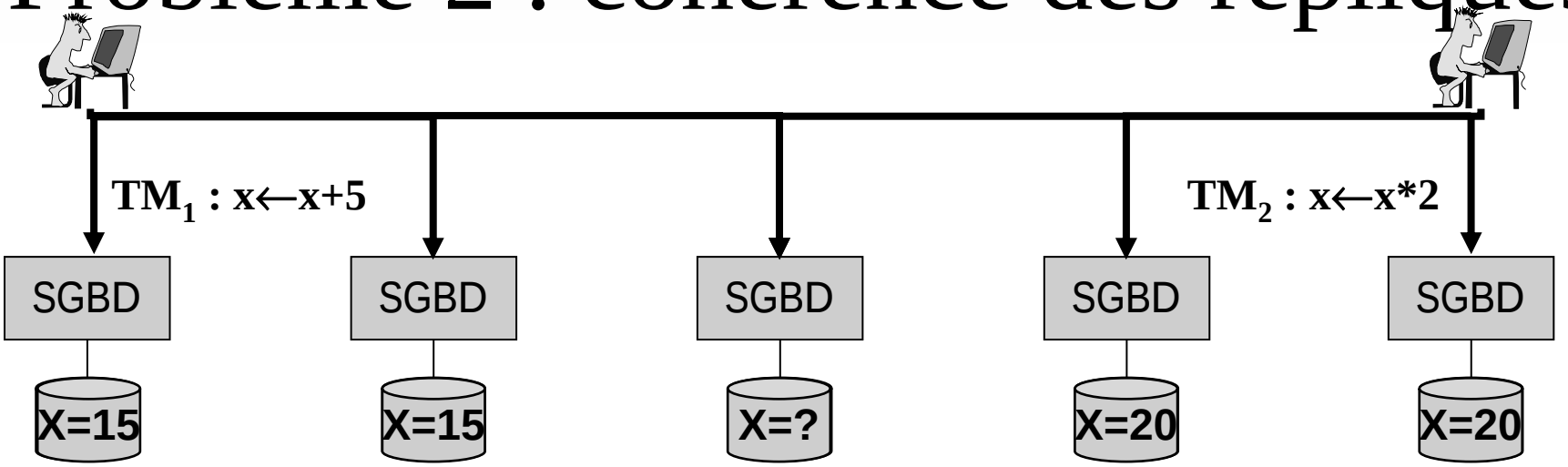
⇒ *Minimiser les délais d'attente des transactions*

Problème 2 : cohérence des répliques



- Cohérence mutuelle des copies ?
  - Les mises à jour doivent être propagées d'un nœud à l'autre.
  - La réplication synchrone garantit la cohérence mutuelle des copies mais ralentit la validation des transactions
  - La réplication asynchrone accélère la validation des transactions de mise à jour mais les copies ne sont pas toujours mutuellement cohérentes

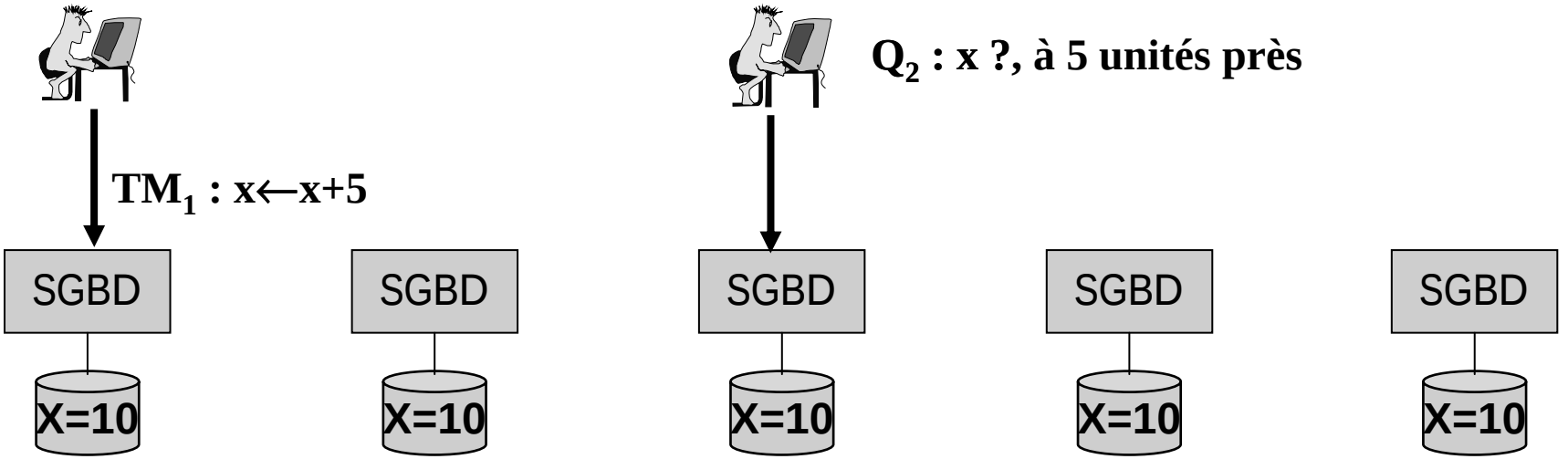
Problème 2 : cohérence des répliques



X=30 ou X=25 selon l'ordre de propagation

- Ordonner les transaction concurrentes en fonction des conflits sur les données
  - Deux transactions qui touchent la même donnée, dont une la modifie, sont exécutées sur tous les nœuds dans le même ordre
  - Deux transactions qui ne touchent pas les mêmes données peuvent être exécutées en parallèle dans n'importe quel ordre
- ⇒ *Les transactions sont exécutées sur tous les nœuds*

Problème 3 : contrôle de la qualité

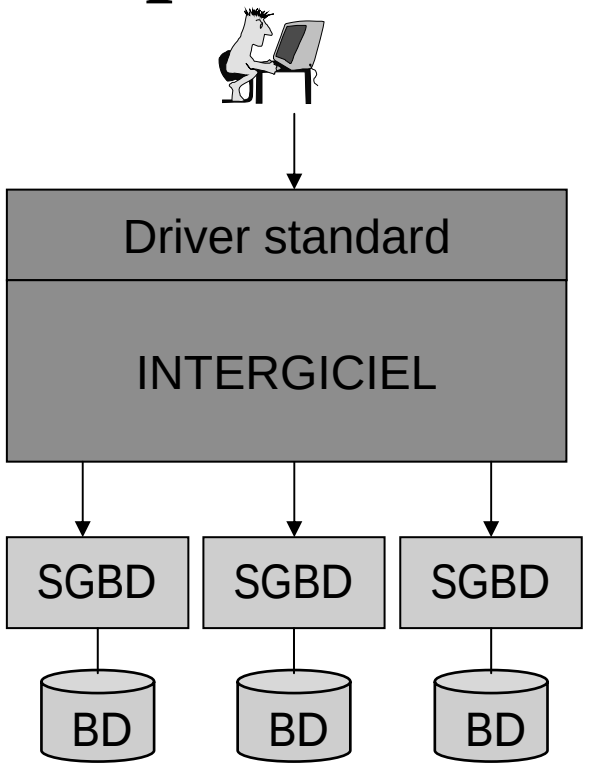


- Les requêtes doivent-elles lire des données parfaitement cohérentes ?
  - Délai d'attente de propagation et de validation
- De nombreuses applications tolèrent des lectures imprécises
  - Statistiques sur entrepôts de données, miroirs de sites Web
  - La qualité d'une lecture est l'erreur de mesure. Elle doit être spécifiée et contrôlée.

Problème 4: réplication transparente

- Préserver l'autonomie de l'application
  - Coûts de migration
  - Simplicité et flexibilité de la conception
- Préserver l'autonomie du SGBD
  - Solution générique
  - Utilisable même avec des SGBD propriétaires
  - Coûts de la modification d'un SGBD
  - Confiance du client

Problème 4: réplication transparente



Code applicatif inchangé

SGBD = boîte noire

Données inchangées

- Solution sous la forme d'une couche intergicielle qui s'intercale entre l'application et le SGBD
- Interface de driver standard d'accès aux données
- *⇒ Gestion de la réplication entièrement gérée par l'intergiciel*

Objectifs

- Les performances des applications bases de données répliquées sur une grappe sont améliorées :
  - en contrôlant la qualité des données disponibles sur la grappe,
  - en effectuant un équilibrage de charge qui tient compte de la qualité, et
  - en adaptant la stratégie de propagation des mises à jour à la charge applicative.

Contributions

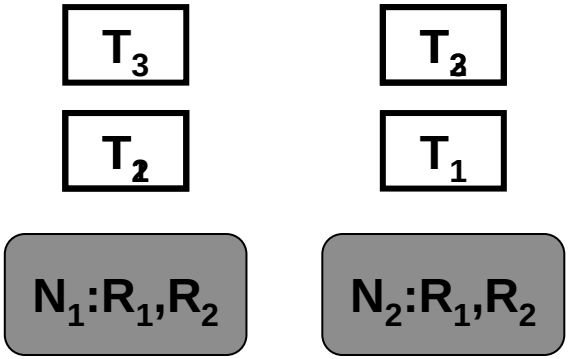
- Contrat de qualité et algorithme générique d'évaluation de la qualité
- Algorithme de routage qui inclut le coût pour obtenir la qualité désirée
- Stratégies de rafraîchissement paramétrables
- Validation expérimentale et intégration dans le projet Leg@net

Contrôle de Qualité

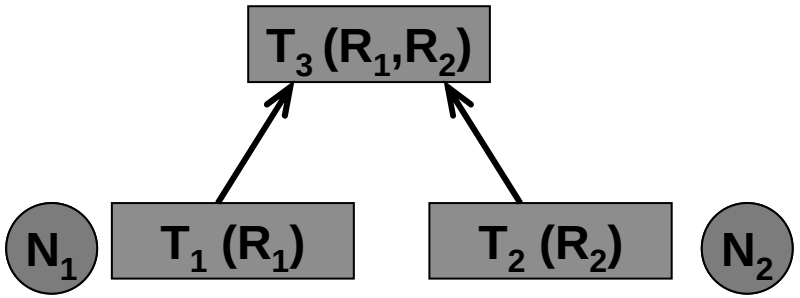
- Définir la notion de qualité de données relationnelles répliquées
  - Définition qualitative : fraîcheur et validité
  - Définition quantitative : mesures de divergence
- Proposer un modèle de contrat de qualité flexible
  - Différentes mesures
  - Différentes granularités
- Proposer un algorithme d'évaluation
  - Indépendant des mesures
  - Non intrusif

Précédence des transactions

- Ordre de précédence
  - T $\rightarrow$ T ssi T et T' sont conflictuelles et que T est plus jeune que T
  - Les transactions sont effectuées sur tous les nœuds dans un ordre compatible avec l'ordre de précédence
- Graphe de précédence
  - Stocke l'ordre d'exécution des transactions et l'état des nœuds de la grappe



grappe des noeuds



graphe de précédence

Définition qualitative de la qualité

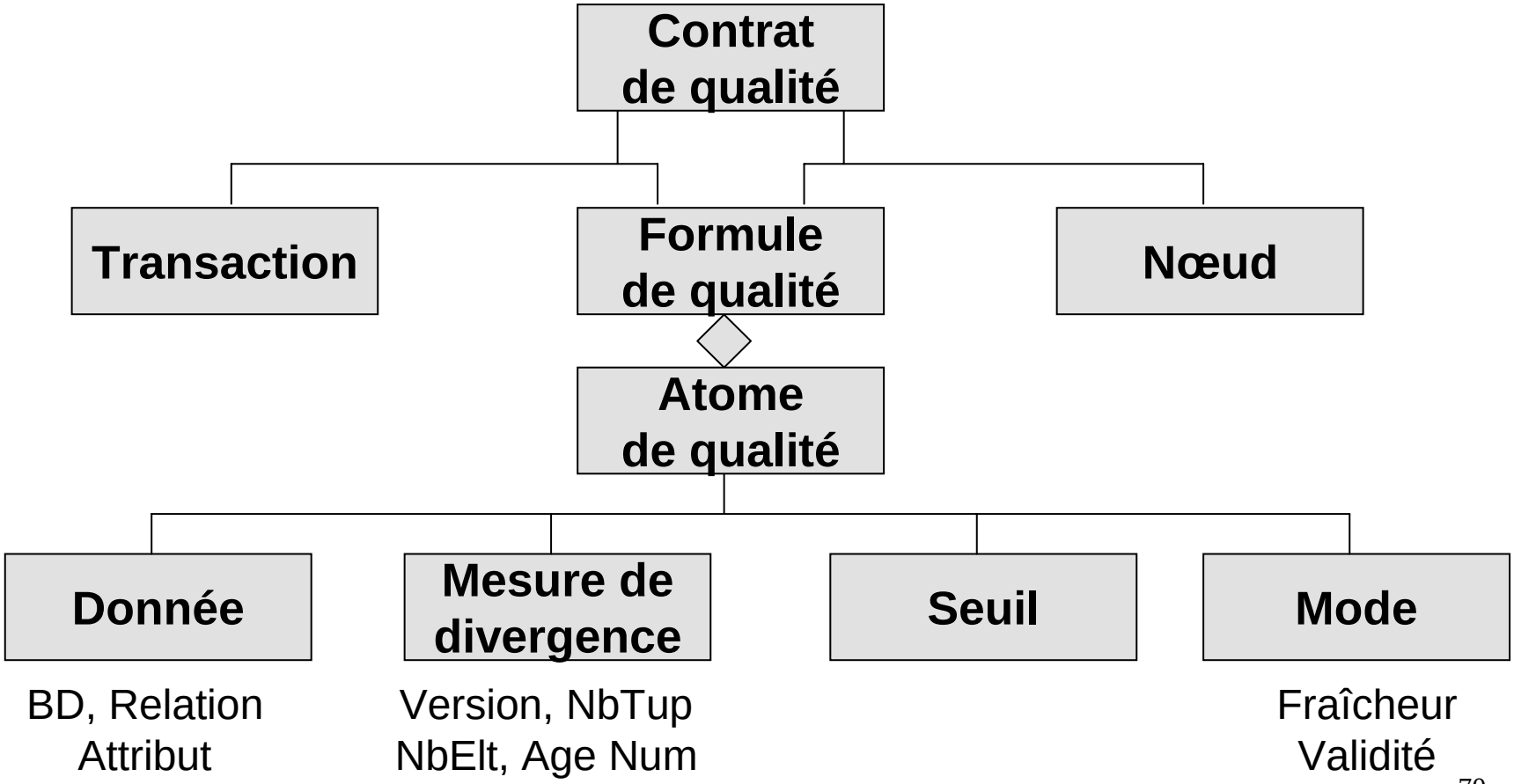
- Modèle des données
  - Données relationnelles (BD, tables, attributs, éléments)
  - Donnée logique *a*, copie physique a_i sur un nœud N_i
- Notion de qualité
  - Validité : a_i est valide si elle ne reflète que des transactions validées et dont l'exécution sur N_i respecte l'ordre global de précédence
  - Fraîcheur : a_i est fraîche si elle reflète toutes les transactions validées et dont l'exécution sur N_i respecte l'ordre global de précédence
  - Qualité parfaite : a_i est de qualité parfaite si elle est valide et fraîche
-

Définition quantitative de la qualité

- Comment mesurer la qualité d'une copie a_i sur N_i ?
- Ensemble Att(a_i) des transactions en attente sur a_i
  - Validité : Att(a_i)={TM modifiant a_i, en cours d'exéc. sur N_i}
  - Fraîcheur : Att(a_i)={TM modifiant a_i, commencée sur au moins un nœud mais pas terminée sur N_i}
- Mesures de divergence
  - Version(a_i)= nombre de transactions en attente sur a_i
  - NbTup(a_i) = nombre de tuples non valides ou non frais de a_i
  - NbElt(a_i) = nombre d'éléments non valides ou non frais de a_i
  - Age(a_i) = âge de plus ancienne transaction en attente sur a_i
  - Num(a_i) = distance euclidienne, pour a attribut numérique

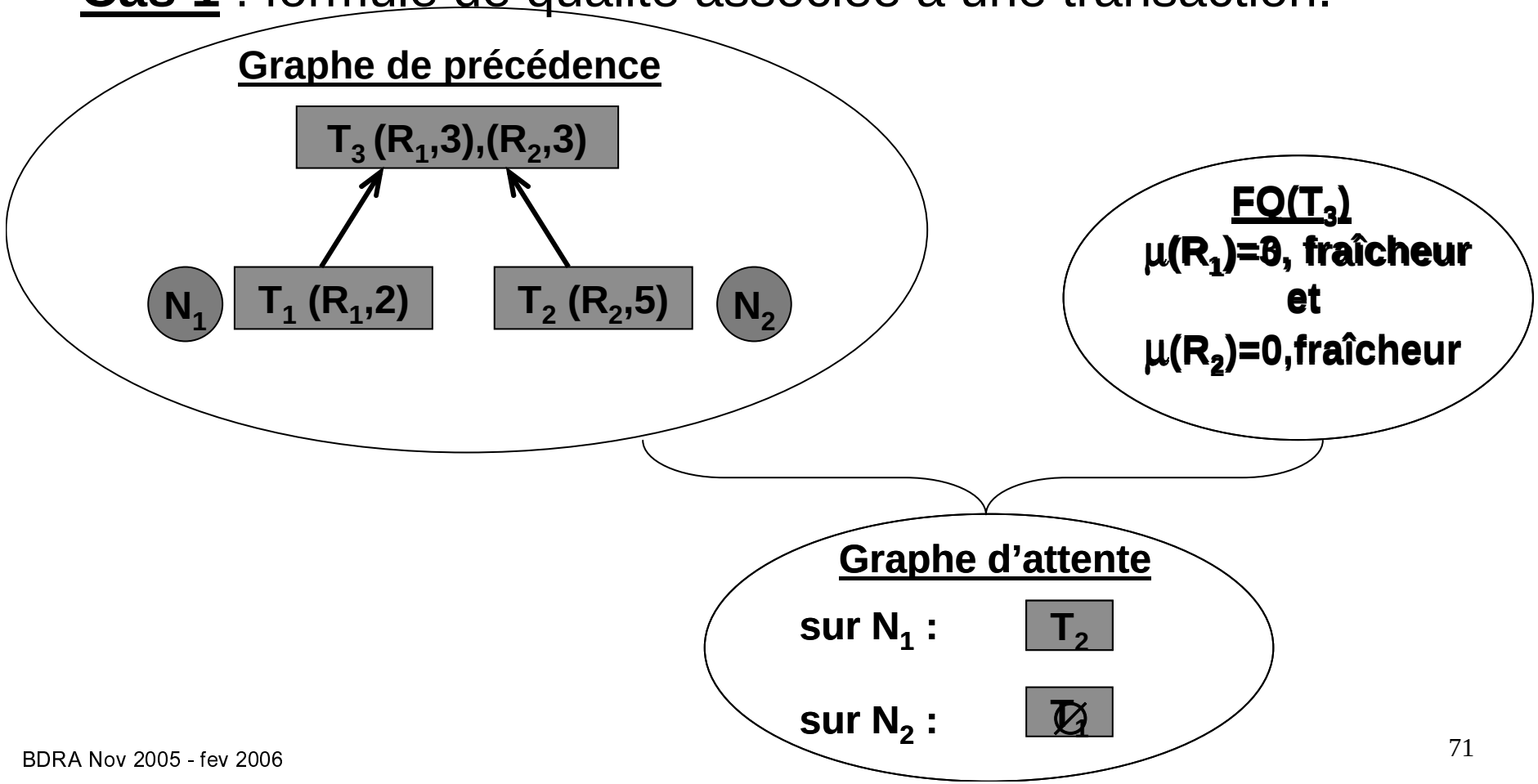
Contrat de qualité

- Permet de spécifier la qualité minimale exigée
  - soit pour l'exécution d'une transaction sur un nœud,
  - soit pour les données d'un nœud de la grappe



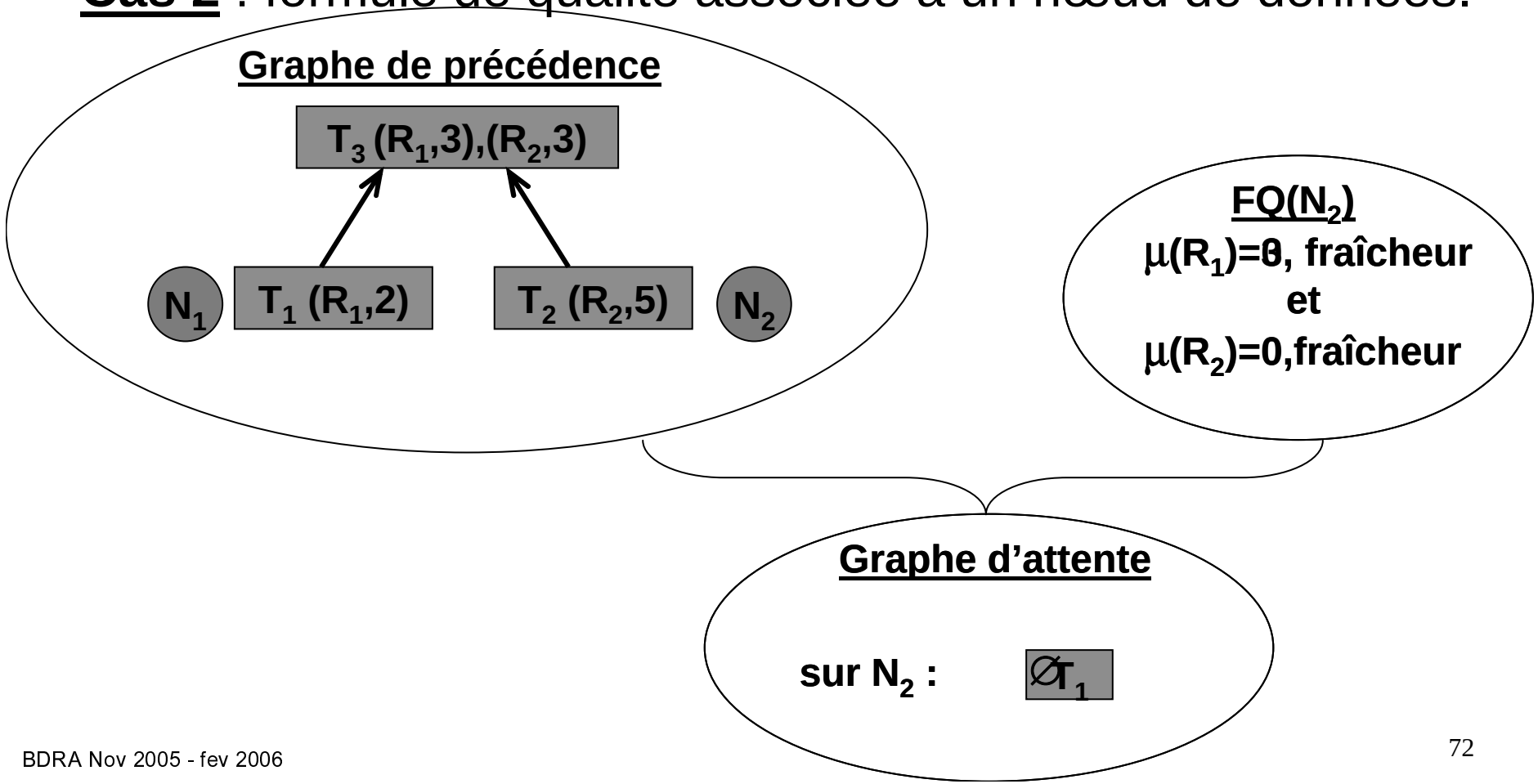
Graphe de refresh

- Graphe des transactions de mise à jour qui doivent être exécutées et validées sur un nœud N_i pour satisfaire une formule de qualité *FQ*.
- **Cas 1** : formule de qualité associée à une transaction.



Graphe de refresh

- Graphe des transactions de mise à jour qui doivent être exécutées et validées sur un nœud N_i pour satisfaire une formule de qualité *FQ*.
- **Cas 2** : formule de qualité associée à un nœud de données.



Algorithme d'évaluation

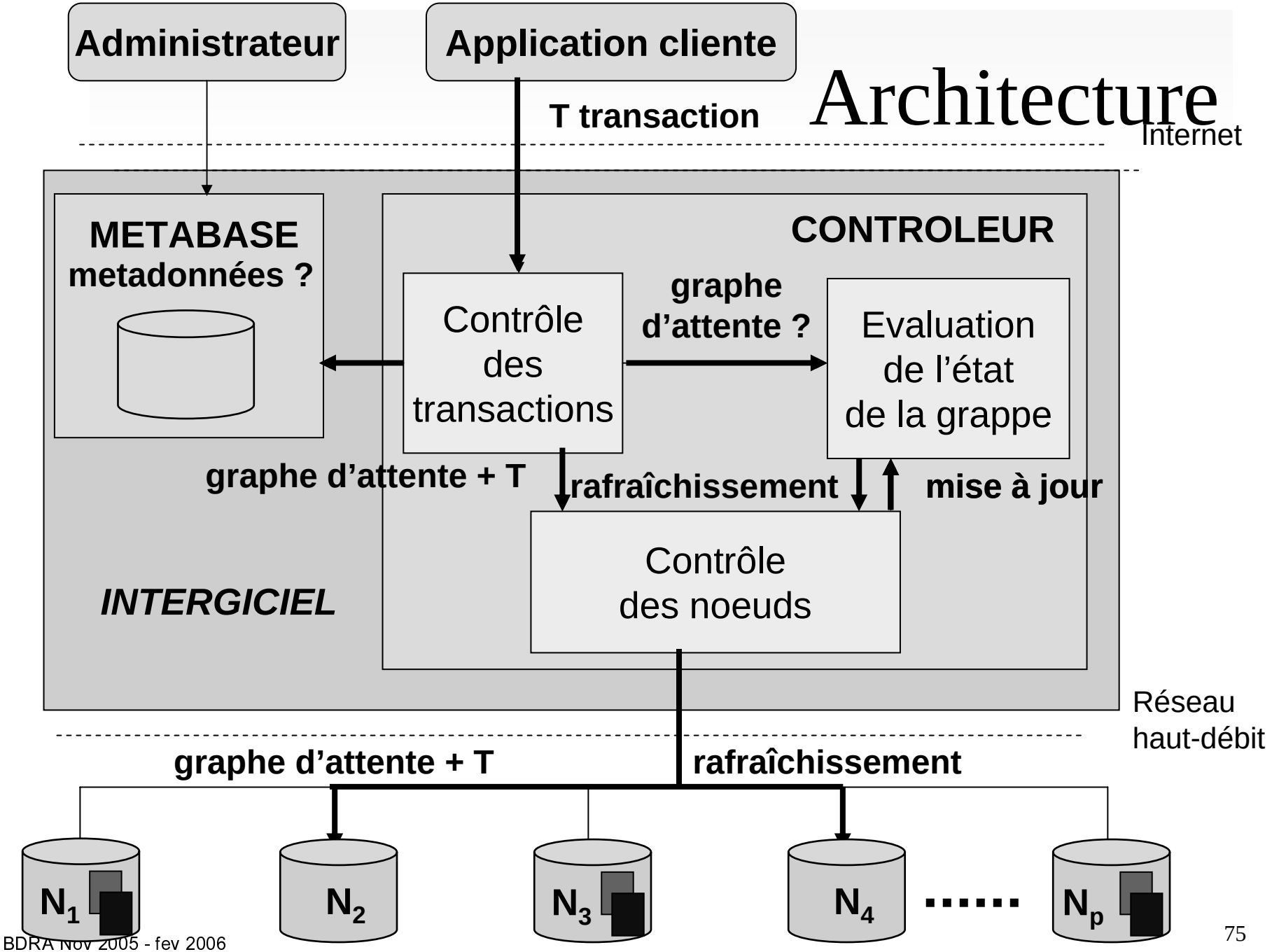
- Pour (a, μ, s, m) un atome et N_i un nœud, calcule le graphe de refresh minimal tel que $\mu(a) < s$, pour le mode m

```
Fonction att_atome(a, μ ,s, m, Ni){
   Tset = ∅; div=0;
   Pour tout T feuille Faire
      Tset = Tset U do_att_atome(T, a, μ, s, m, div, false)
   retourne (Tset,→)
}

Fonction do_att_atome(T, a, μ, s, m, div, encours){
   Nset= ∅; ec=encours;
   si T terminee sur Ni alors retourne Nset; // test de terminaison

  // en mode VALIDITE, attendre la première transaction en cours d'exécution
   si m=VALIDITE et encours=false et T en cours d'exécution sur Ni alors ec=true

   si m=FRAICHEUR ou (m=VALIDITE et ec=true)
      si necessaire(T, a, μ, s, m, div) alors Nset={T};
         pour tout T' parent de T faire
            Nset=Nset U do_att_atome(T', a, μ, s, m, changediv(div, T', a, μ),ec);
   retourne Nset
}
```

Evaluation pratique du graphe d'attente

- Evaluation des données touchées par une transaction
  - DataSet potentiel
  - DataSet réel
- Détection des conflits
  - Conflits potentiels
  - Conflits réels
- Evaluation des modifications d'une transaction de mise à jour
  - Evaluation a priori
  - Evaluation a posteriori

Architecture



Routage des transactions

- Problème: comment choisir le meilleur nœud d'exécution pour une transaction T ?
  - coût(T, N_i) = coût_ref(T) + coût_rafraîchissement(T, FQ) + charge(N_i)
  - coût_ref(T) = temps d'exécution de référence de T sur nœud vide
  - coût_rafraîchissement(T, FQ) = Σ coût_ref(T_i), T_i en attente de rafraîchissement sur N_i
  - charge(N_i) = Σ temps_restant(T_i), T_i en cours d'exécution sur N_i
- Le routage tient compte de la qualité demandée
  - Dans certains cas, un nœud obsolète mais peu chargé est meilleur qu'un nœud frais mais très chargé.

Stratégies de rafraîchissement

- Problème : comment propager les mises à jour entre les nœuds de la grappe ?
- Stratégies de rafraîchissement
  - Stratégie ::= ({Evénement}, Destination, Quantité)
  - Evénement ::=
    - Routage(T, N_j)
    - Souscharge(N_j, min)
    - Obsolète(N_j,FQ)
    - Validation(T, N_j)
    - Période(t)
  - Destination ::= { Noeud }
  - Quantité ::= NbMax | ChargeMax | Cst | QualitéMin(FQ)

Exemples de stratégies de rafraîchissement

- Stratégie dépendante du routage
  - A la demande (ALD)
- Stratégies indépendantes du routage
  - Dès que possible (DQP)
  - Périodique, fonction de la période
  - Contrôle de la charge, fonction de la charge maximale tolérée
  - Contrôle de la qualité, fonction de la qualité minimale tolérée
- Stratégies hybrides
  - Chaque stratégie indépendante du routage possède une version hybride combinée avec la stratégie de rafraîchissement à la demande

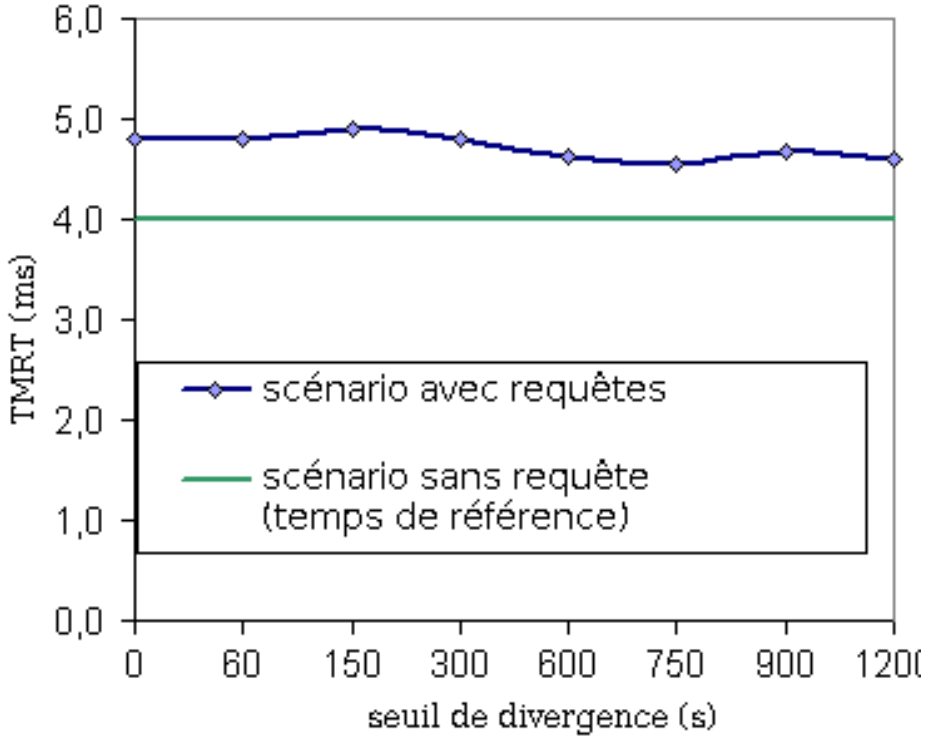
le prototype REFRESCO

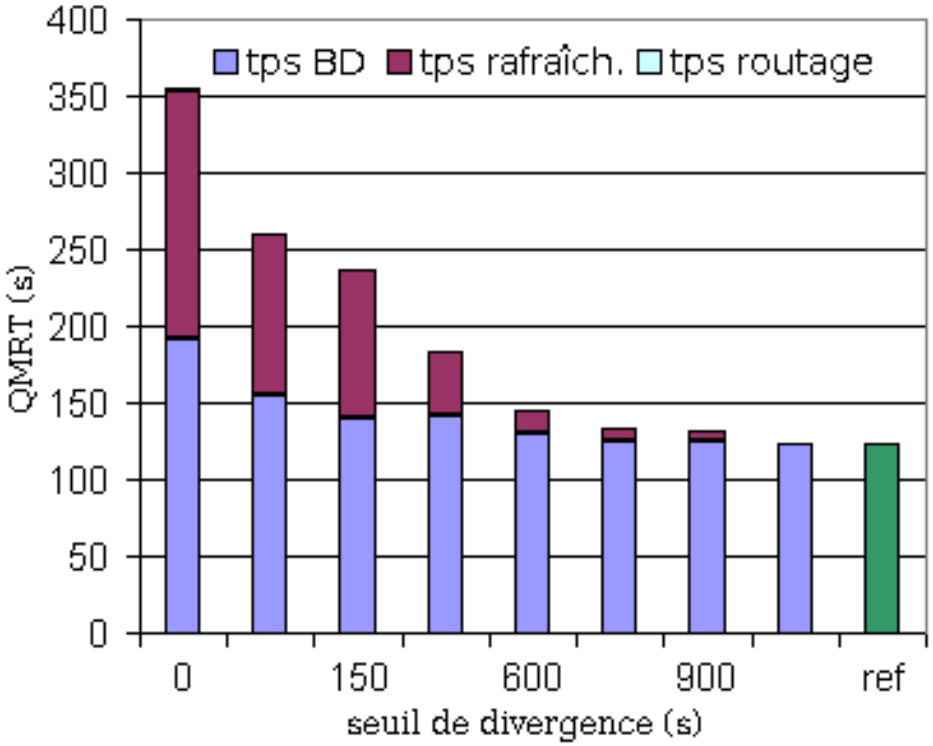
- Configuration monomaître
- Environnement expérimental
  1. Dans un environnement réel (5/64 nœuds, Oracle/Postgresql, banc d'essai de TPC-R, intégration projet Leg@net)
  2. Par simulation (128 nœuds simulés, Simjava)
- Modèle de charge applicative

  (Charge trans., Charge requ, taux de conflit, div. tolérée)
- Mesures de performances
  - Débit, temps moyen de réponse

REFRESCO : validation (1)

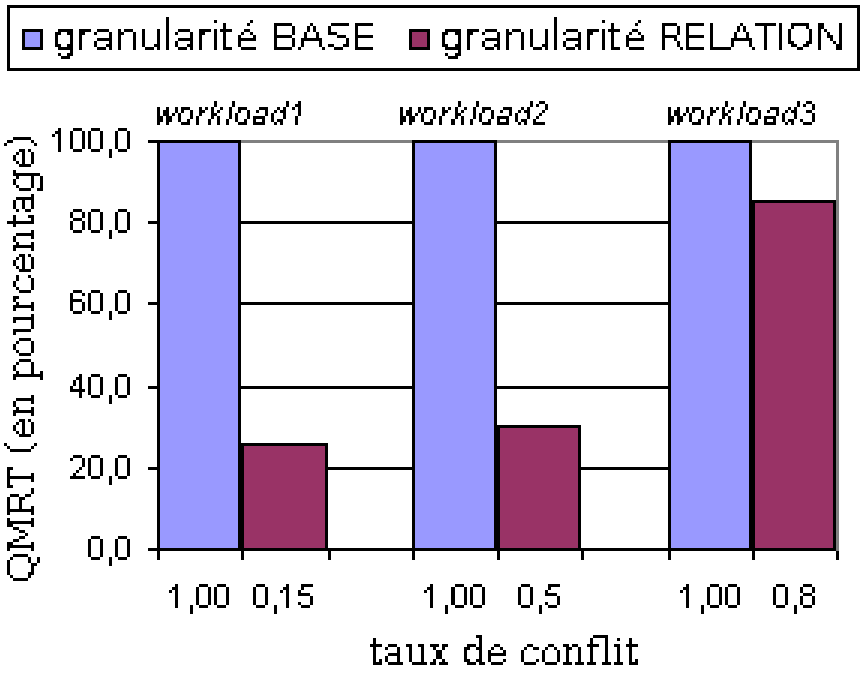
- Impact du seuil de divergence (mesure Age)

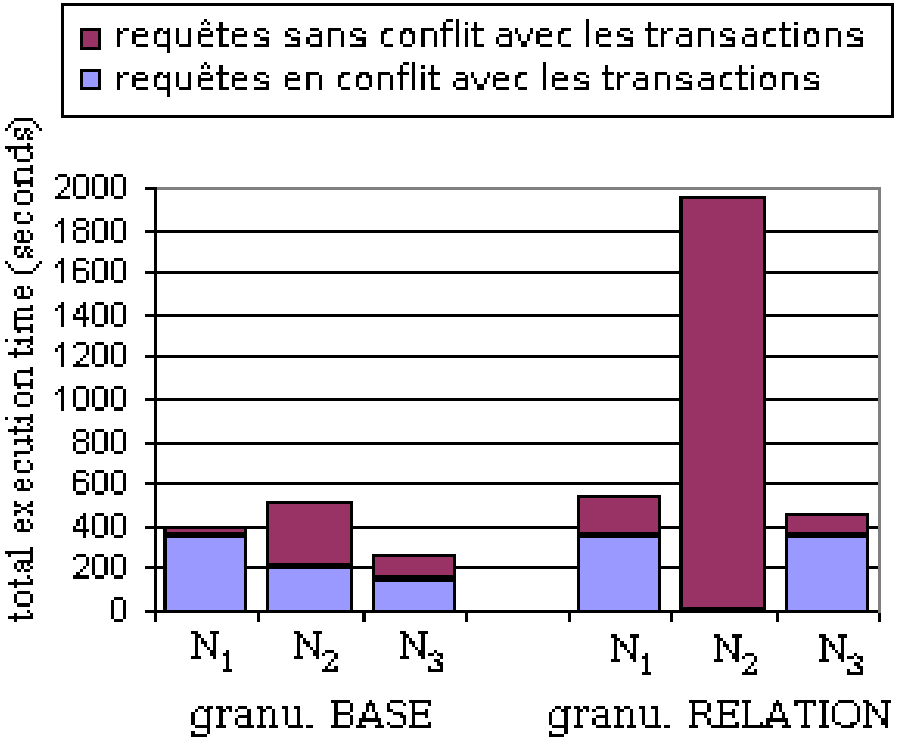




REFRESCO : validation (2)

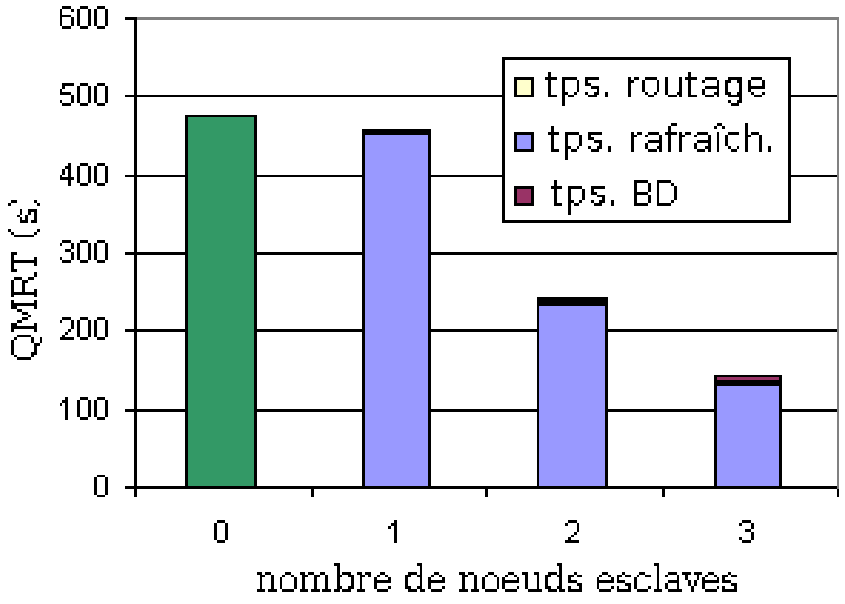
- Impact de la granularité (mesure Age)

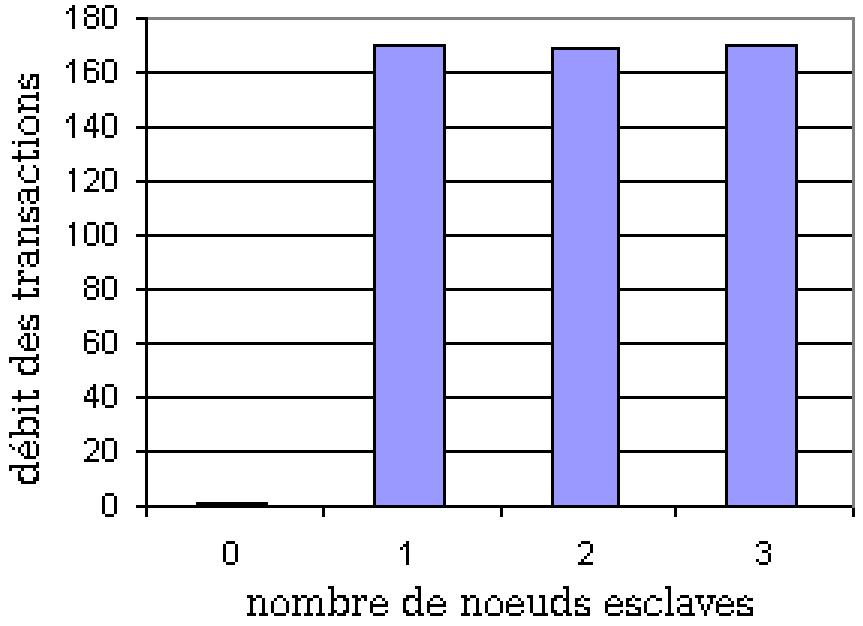




REFRESCO : validation (3)

- Impact du nombre de nœuds (mesure Age)





Validation : conclusion

- Relâcher la fraîcheur des requêtes permet d'améliorer les temps de réponse des requêtes, surtout si la granularité est fine.
- Dans tous les cas, la stratégie hybridée est au moins aussi rapide que la version de base car la stratégie à la demande n'effectue jamais de rafraîchissement inutile.
- Le choix de la meilleure stratégie de rafraîchissement dépend de la charge applicative et de la qualité demandée.

Perspectives

- Extension des travaux de validation du modèle
  - mesure numérique
  - configuration multimaître ou hybride
  - notion de validité avec SGBD en mode « Read uncommitted »
  - évaluation de la charge avec sonde système
  - extension du modèle de la charge applicative
- Contexte large-échelle / Web / Pair à Pair
  - Projet ARA Respire
  - Stages de M2 (voir site)