NEO: BDD objet distribuée transactionnelle

Points abordés:

- Une BDD objet: ZODB
- Vue d'ensemble de NEO
- Répartition des données
- Transactions et parallélisme
- Reprise sur sinistre

Nexedi
Enterprise
Object

Soutenez NEO !
www.neoppod.org

Une BDD objet: ZODB

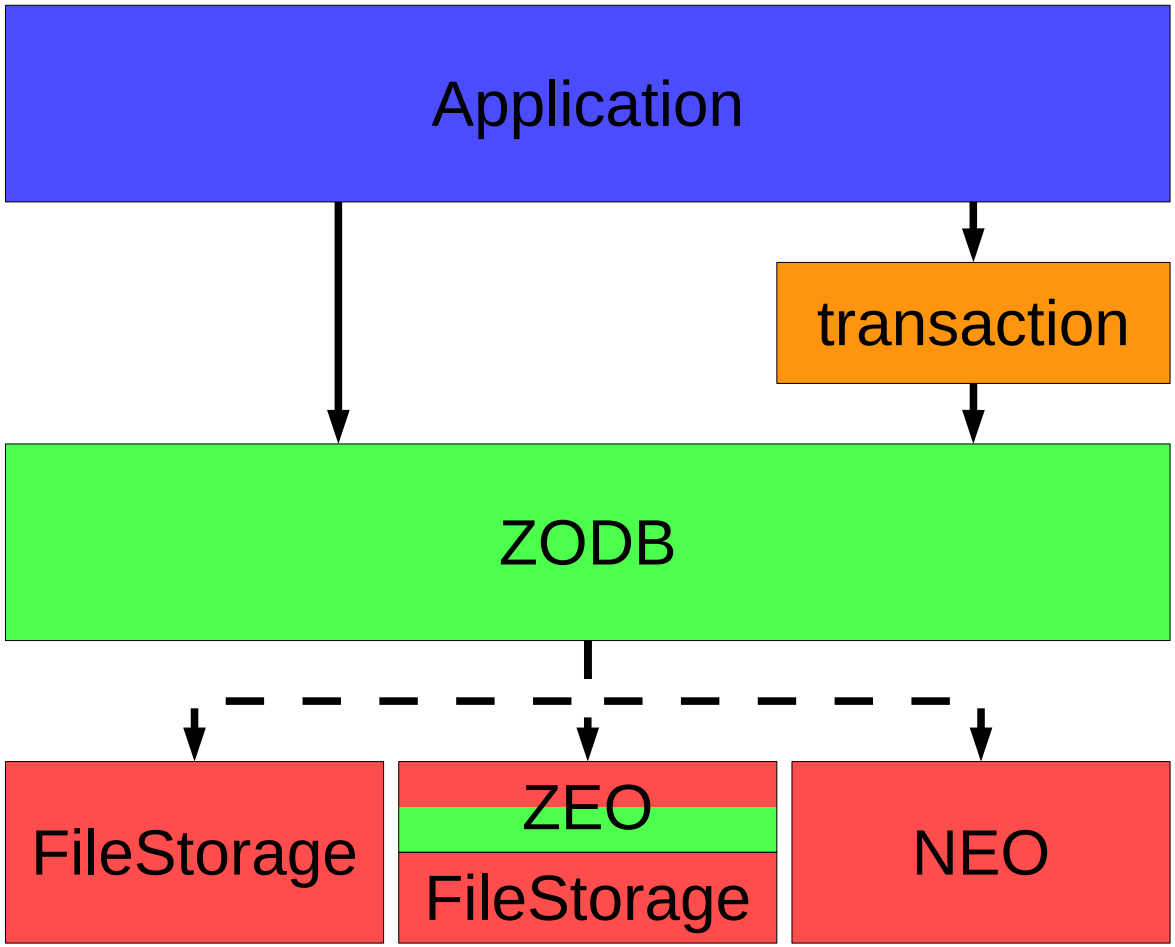
NEO

Répartition des données

Transactions et parallélisme

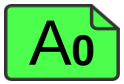
Reprise sur sinistre

Environnement



Fonctionnalités

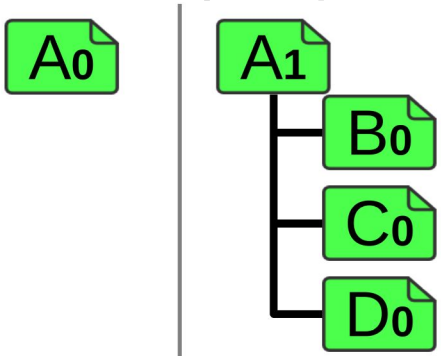
Vue hiérarchique (niveau ZODB)



Vue objet (niveau ZODB.Storage)

Fonctionnalités

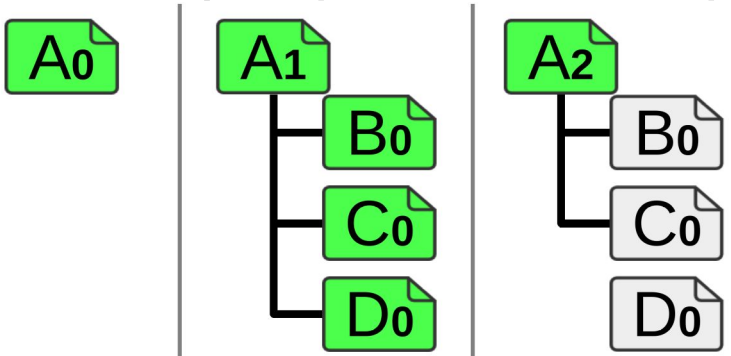
Vue hiérarchique (niveau ZODB)



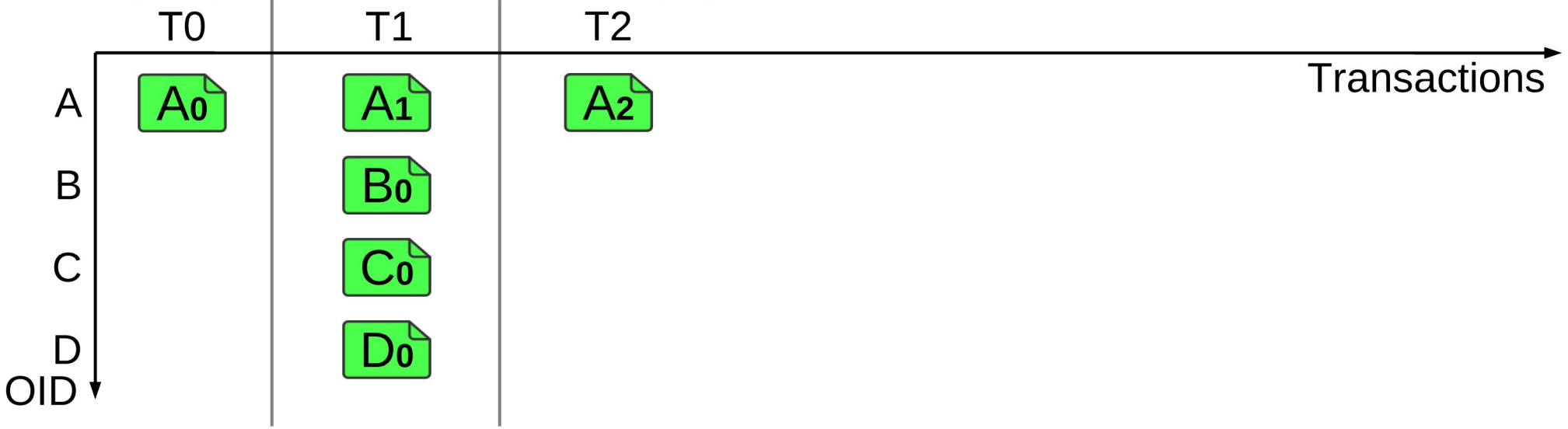
Vue objet (niveau ZODB.Storage)

Fonctionnalités

Vue hiérarchique (niveau ZODB)

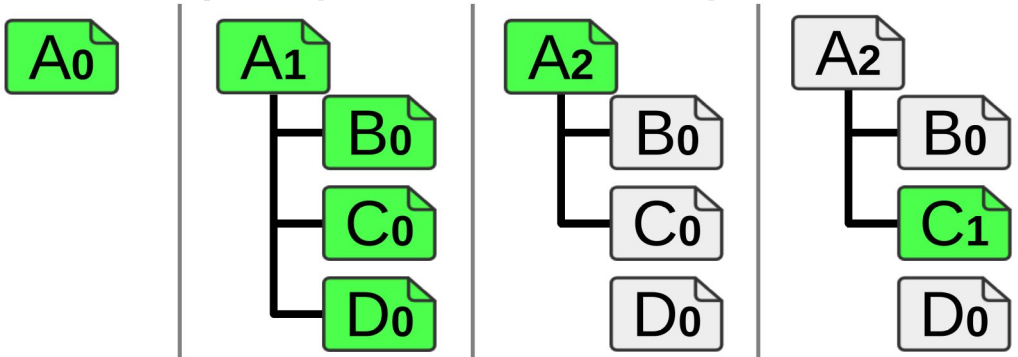


Vue objet (niveau ZODB.Storage)

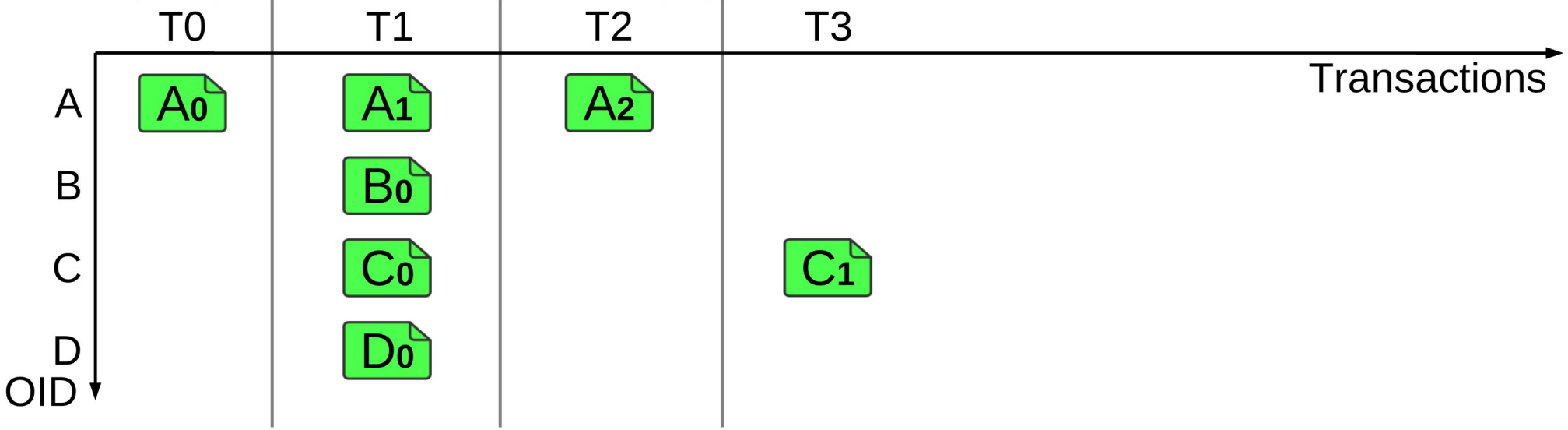


Fonctionnalités

Vue hiérarchique (niveau ZODB)

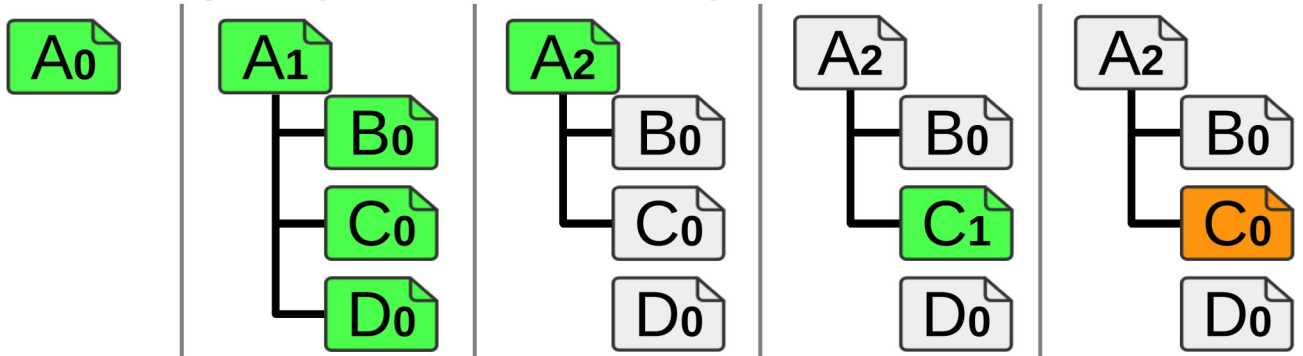


Vue objet (niveau ZODB.Storage)

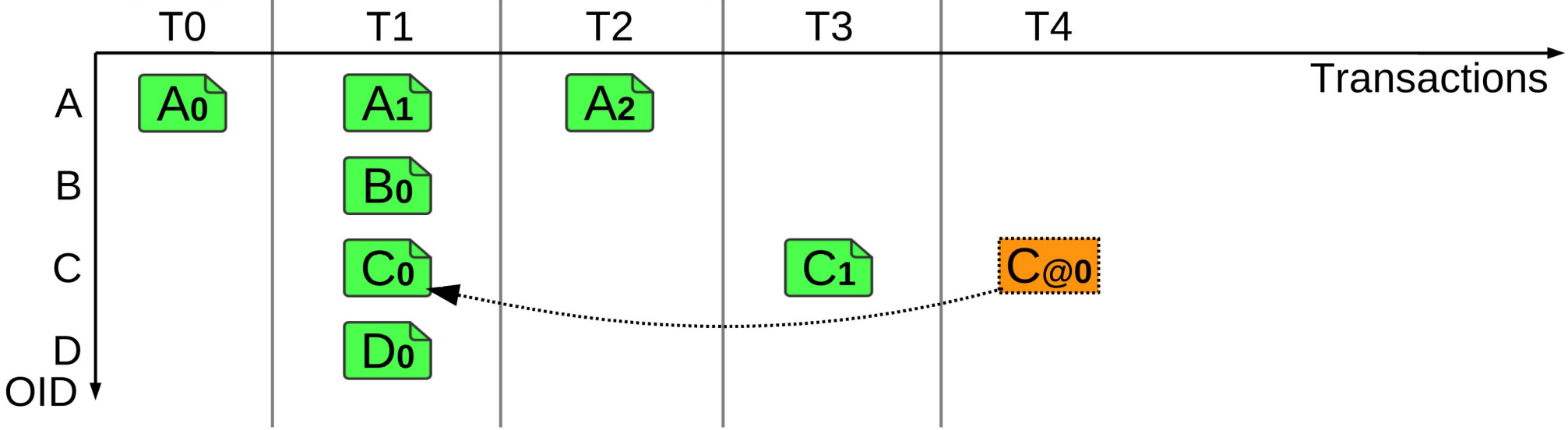


Fonctionnalités

Vue hiérarchique (niveau ZODB)

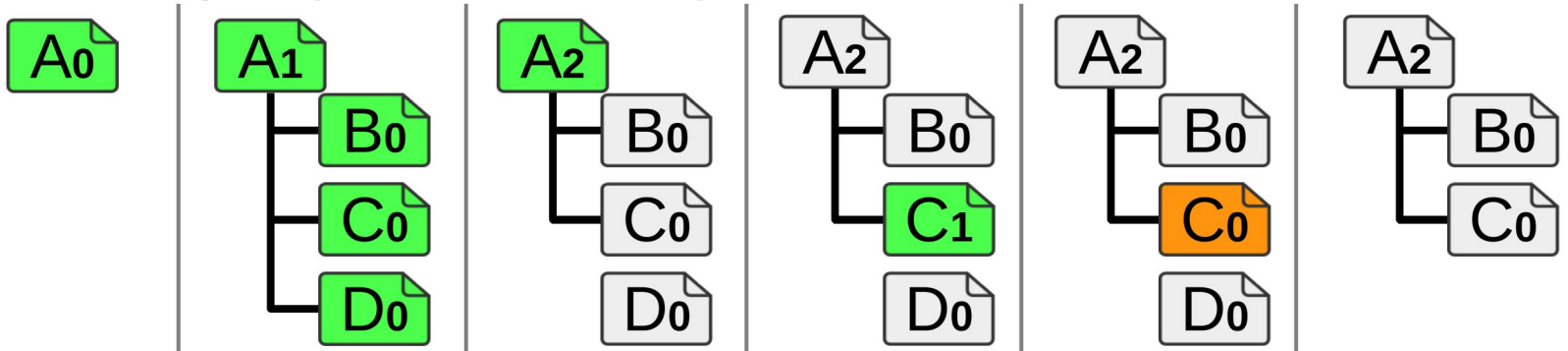


Vue objet (niveau ZODB.Storage)

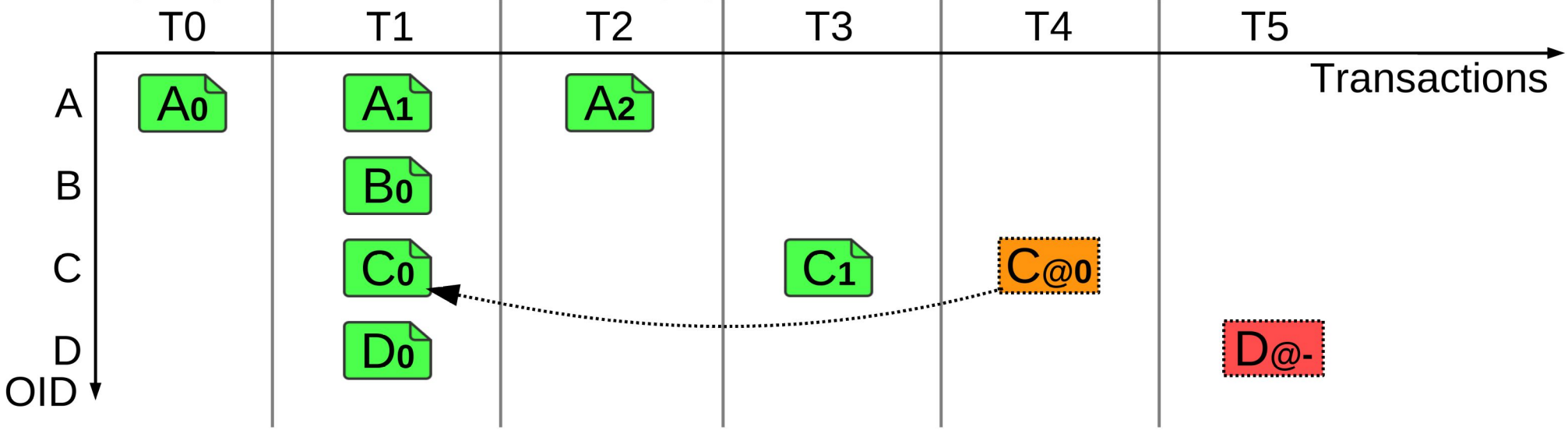


Fonctionnalités

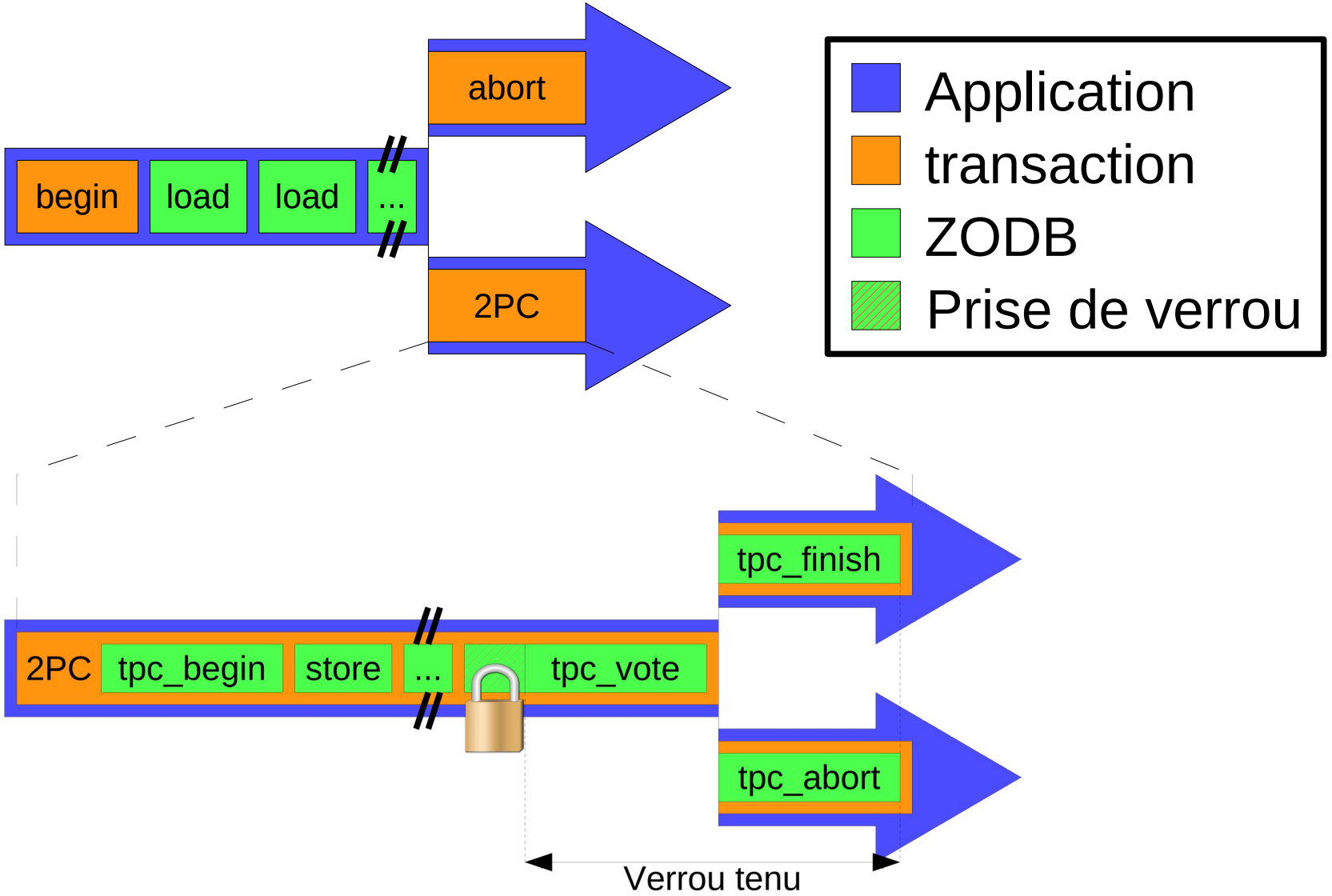
Vue hiérarchique (niveau ZODB)



Vue objet (niveau ZODB.Storage)



Transactions



Une BDD objet: ZODB

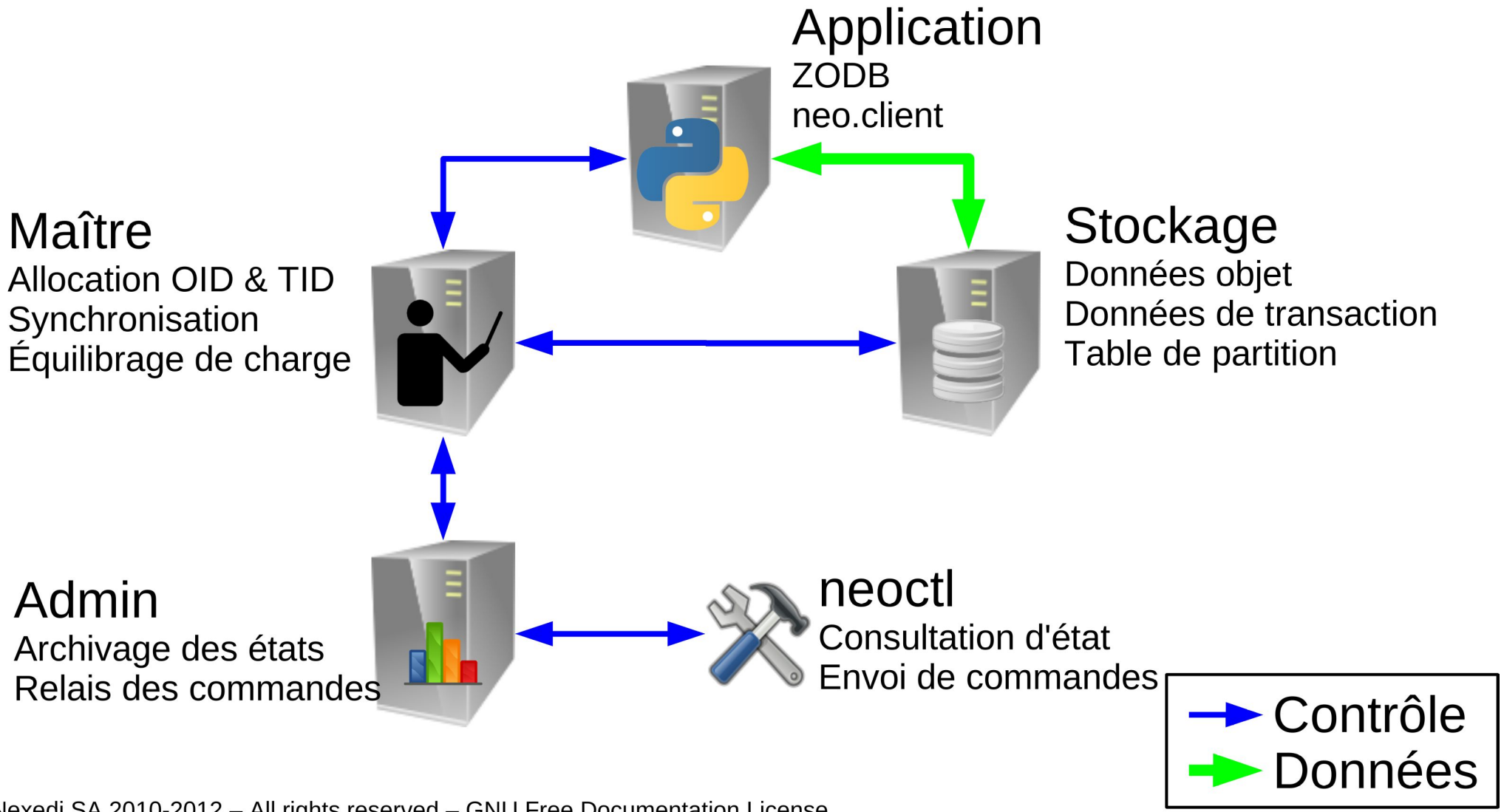
NEO

Répartition des données

Transactions et parallélisme

Reprise sur sinistre

Vue d'ensemble

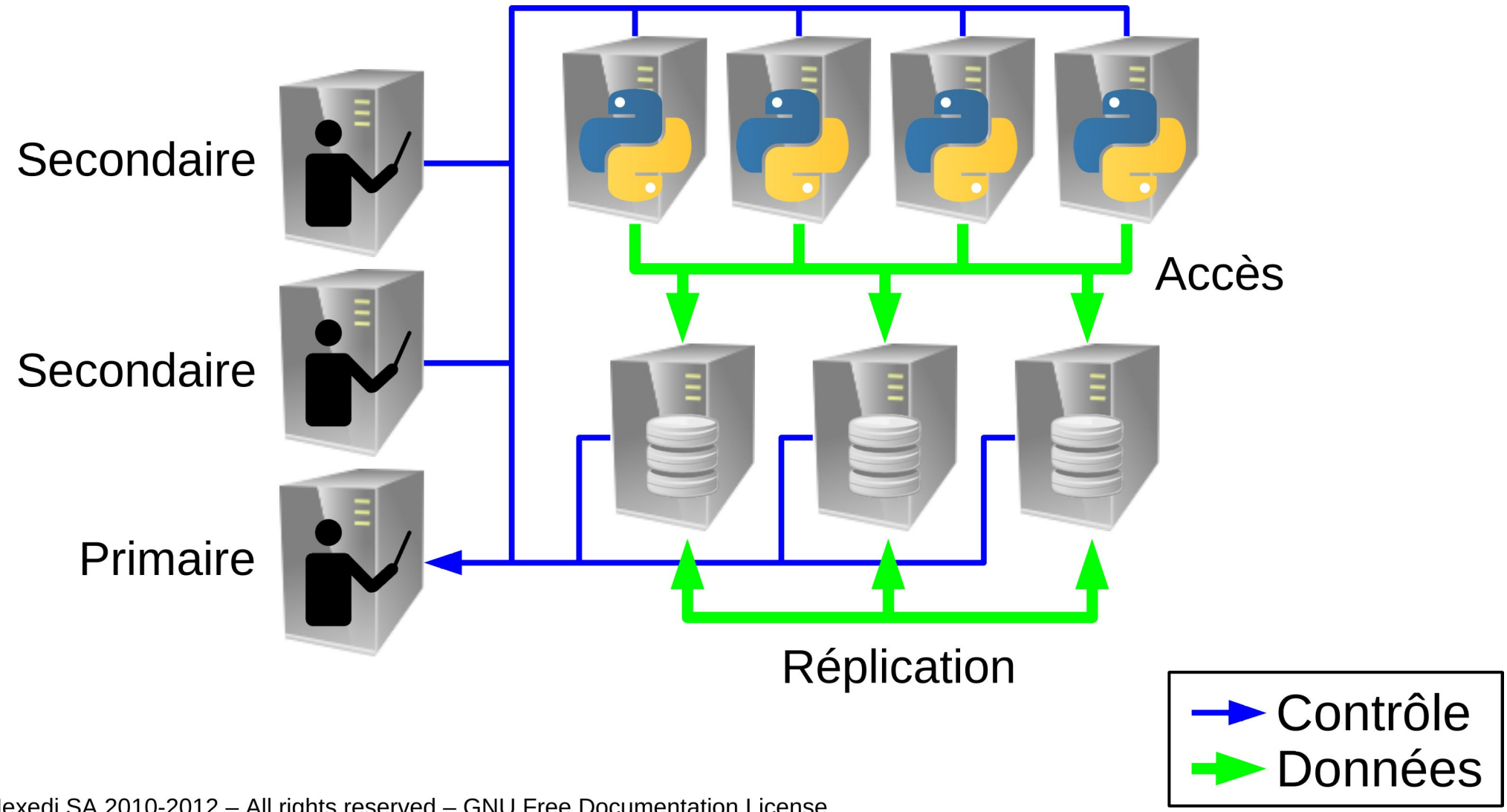


Une BDD objet: ZODB

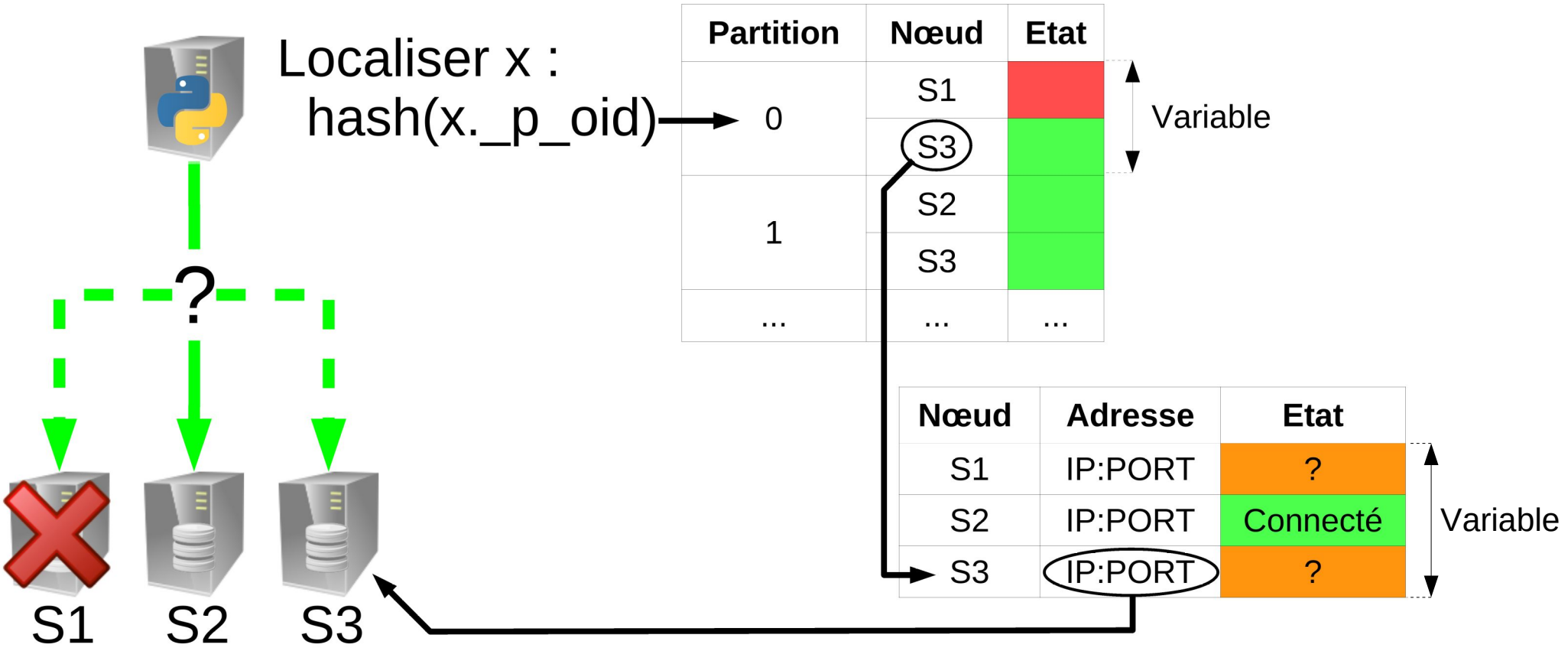
NEO

- Répartition des données
- Transactions et parallélisme
- Reprise sur sinistre

Répartition de charge



Localiser un objet : table de partition



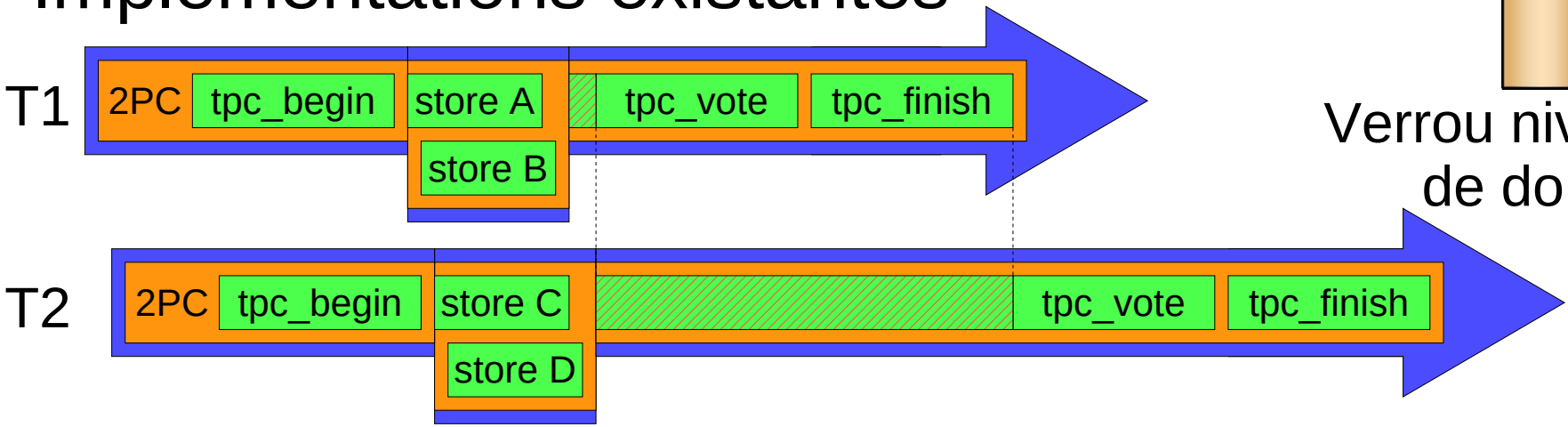
Une BDD objet: ZODB

NEO

- Répartition des données
- Transactions et parallélisme
- Reprise sur sinistre

Verrouillage

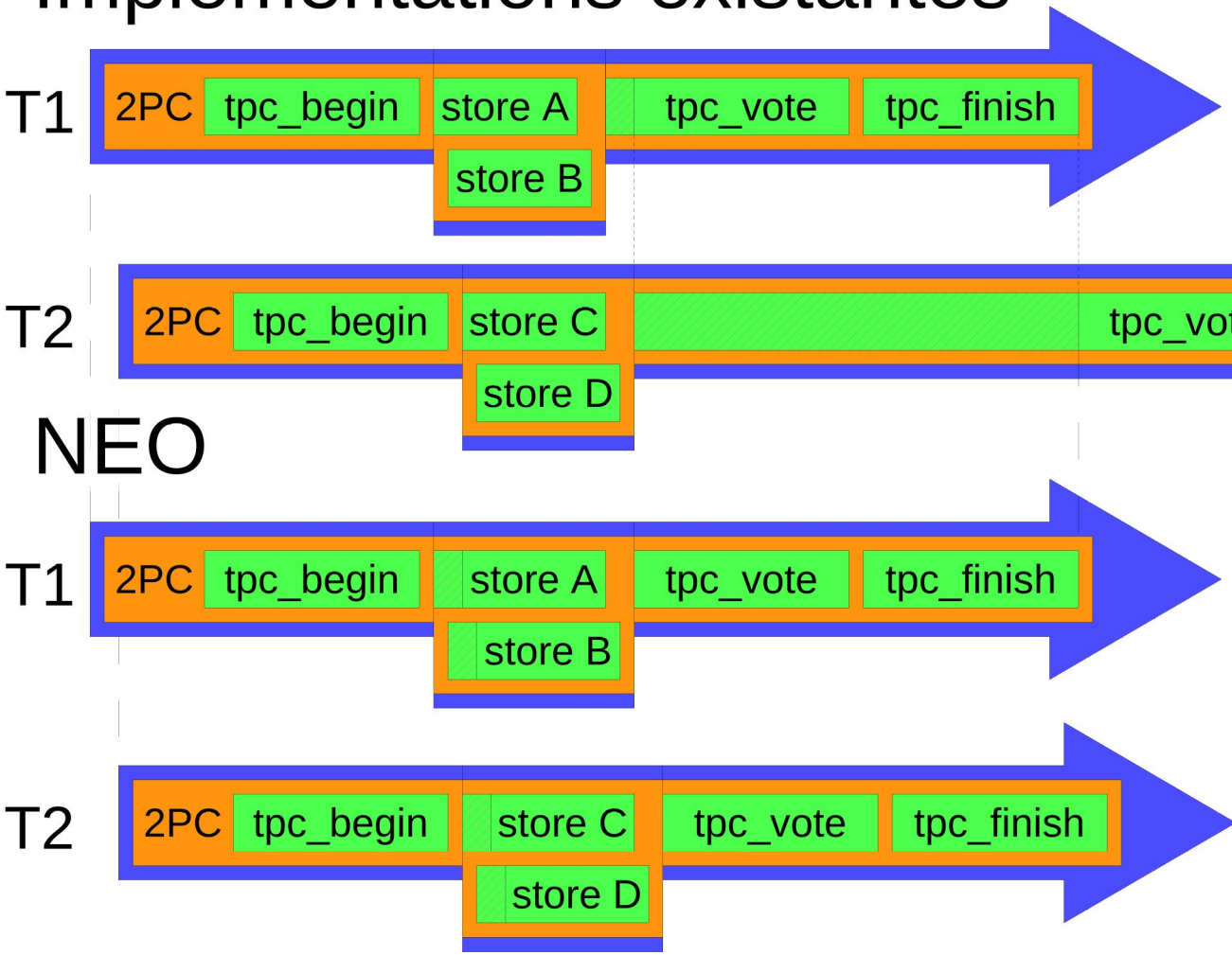
Implémentations existantes



Verrou niveau base de données

Verrouillage

Implémentations existantes



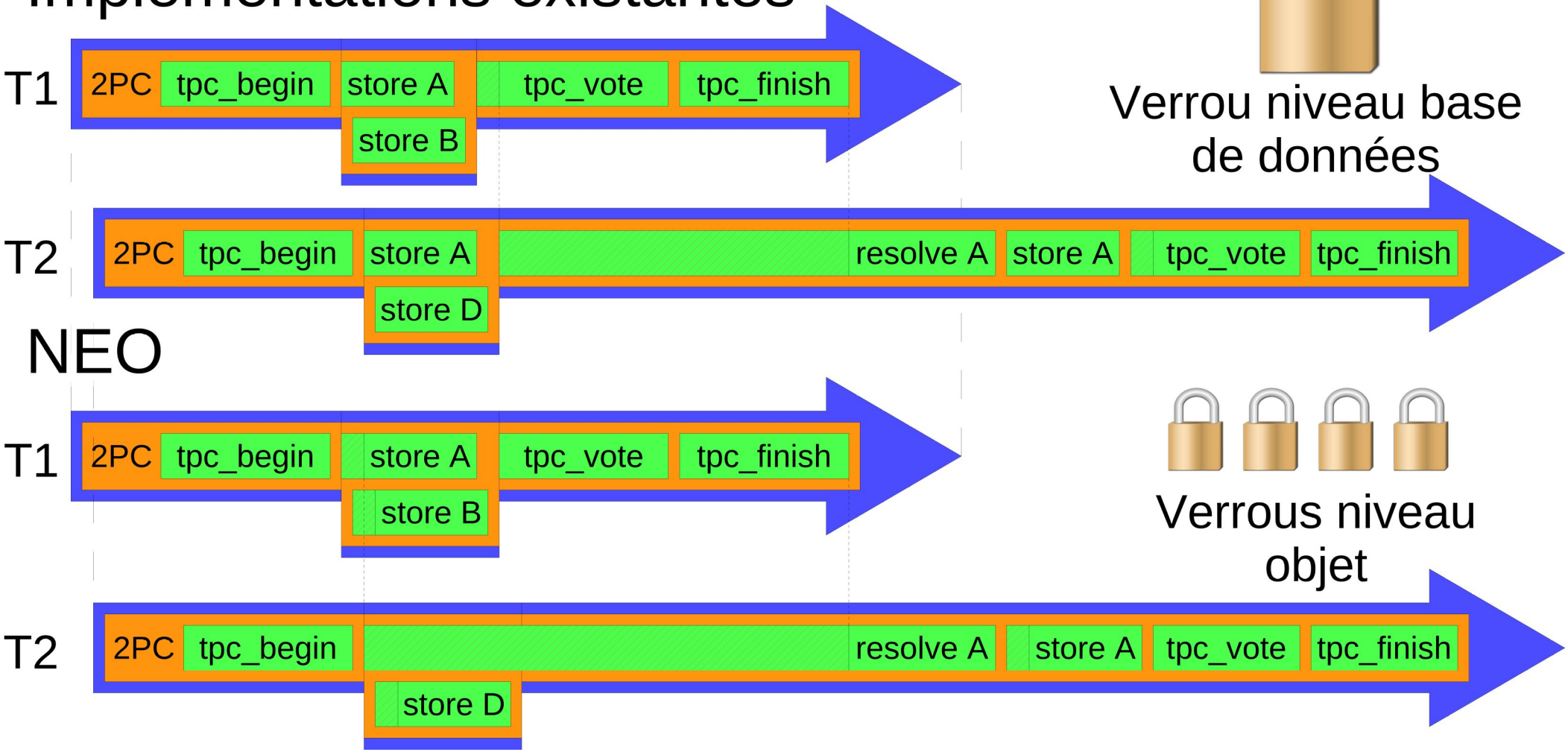
Verrou niveau base de données

NEO

Verrous niveau objet

Verrouillage : cas d'un conflit

Implémentations existantes



Verrou niveau base de données

NEO

Verrous niveau objet

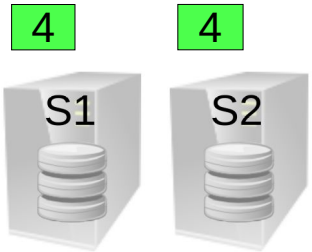
Une BDD objet: ZODB

NEO

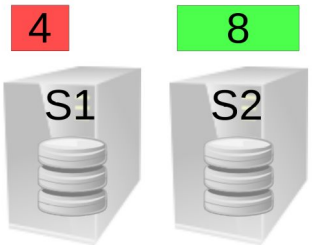
- Répartition des données
- Transactions et parallélisme
- Reprise sur sinistre

Déconnexion : réplication

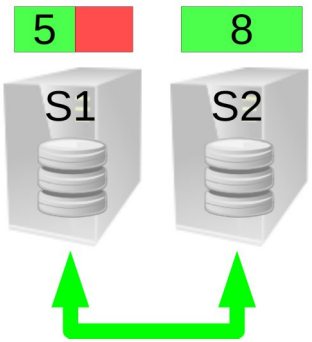
❶ Fonctionnement



❷ Déconnexion

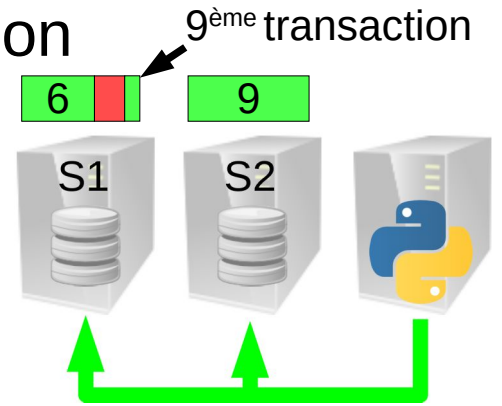


❸ Reconnexion



Partition	Nœud	Etat				
		❶	❷	❸	❹	❺
0	S1					
	S2					
...						

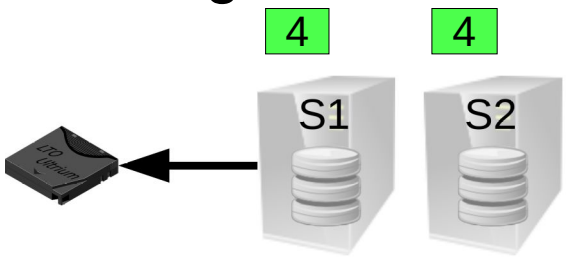
❹ Nouvelle transaction en cours de réplication



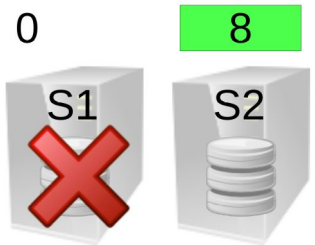
❺ Retour à la normale -> ❶

Sinistre localisé : sauvegardes

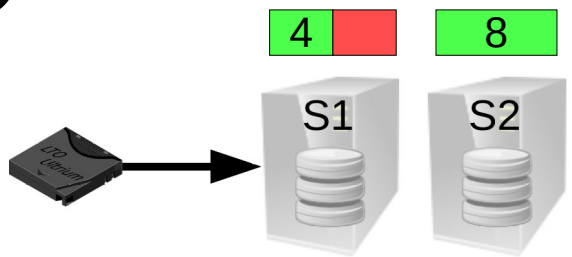
❶ Sauvegarde



❷ Perte de données

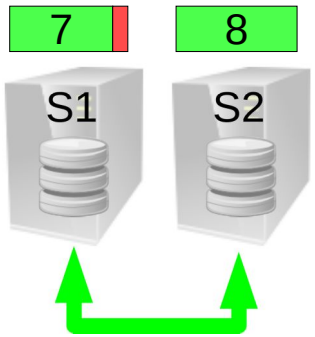


❸ Restauration



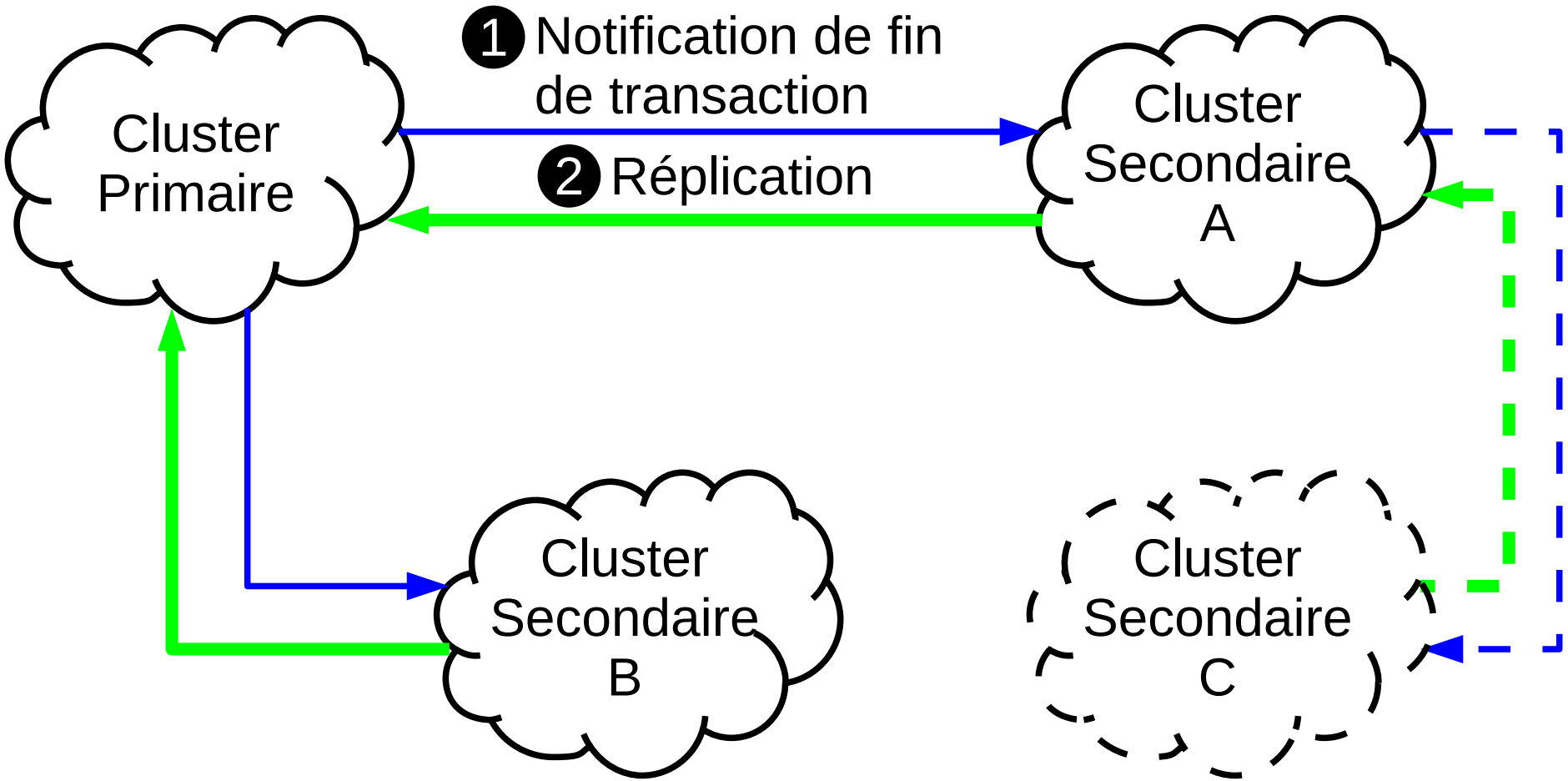
Partition	Nœud	Etat				
		❶	❷	❸	❹	❺
0	S1					
	S2					
...						

❹ Réplication



❺ Retour à la normale -> ❶

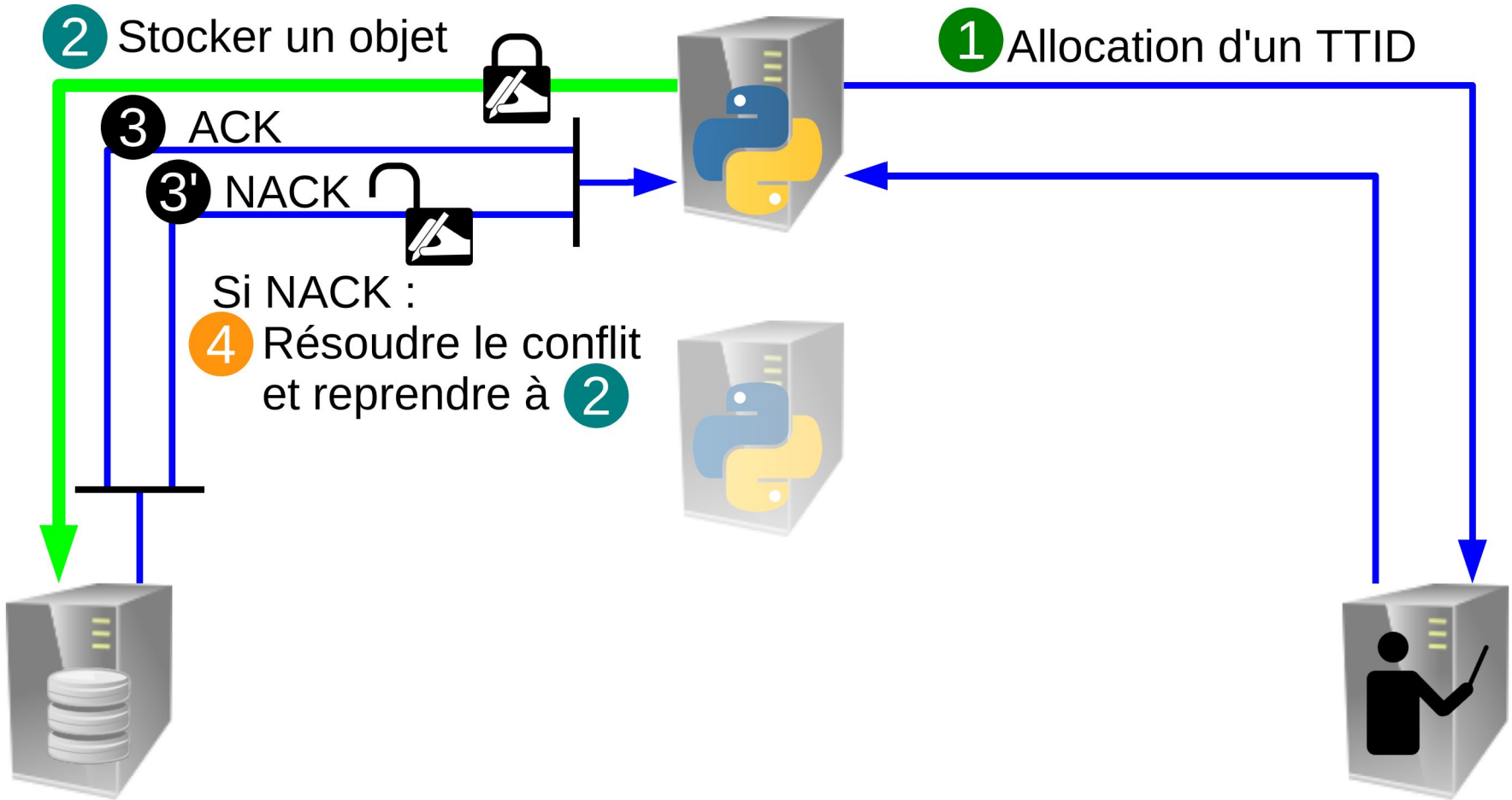
Sinistre étendu : réplication asynchrone



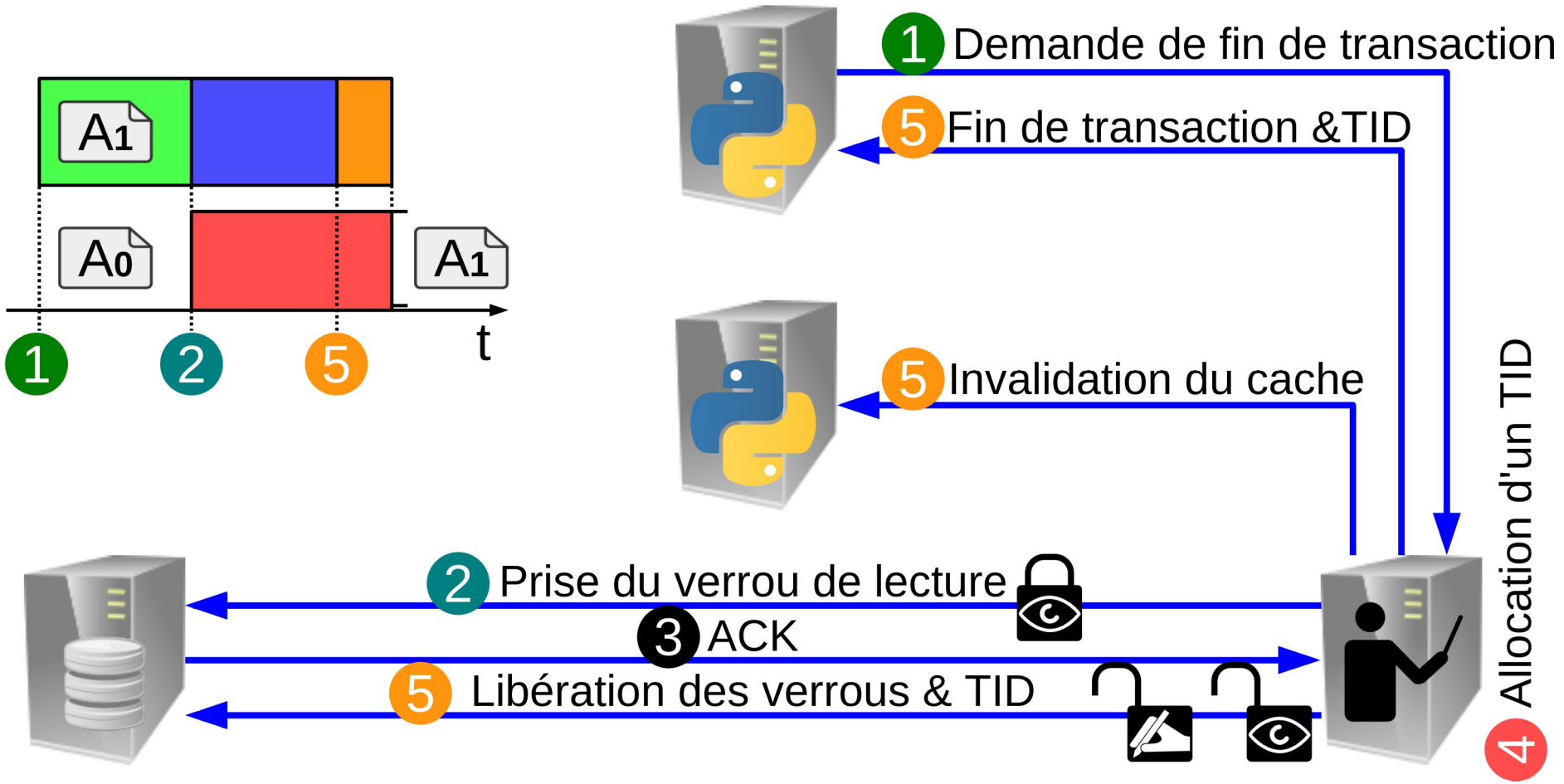
Questions

Soutenez NEO !
www.neoppod.org

Commit en deux phases : phase 1



Commit en deux phases : phase 2



NEO: BDD objet distribuée transactionnelle

Points abordés:

- Une BDD objet: ZODB
- Vue d'ensemble de NEO
- Répartition des données
- Transactions et parallélisme
- Reprise sur sinistre

Nexedi
Enterprise
Object

Soutenez NEO !
www.neoppod.org

NEO (Nexedi Enterprise Objects) est une implémentation de l'interface ZODB.Storage visant la mise à l'échelle:

- grands volumes de données (peta-octets)

- haute disponibilité (tolérance de pannes)

- hautes performances (clustering)

Ce projet a été développé par Nexedi, en collaboration avec laboratoires LIP6 et LIPN dont la tâche était de prouver formellement le design de NEO, ainsi que Pilot Systems et MINES ParisTech dont la tâche était de valider par la pratique l'implémentation de NEO en développant et déployant des applications l'utilisant.

Ce document a été présenté à LinuxExpo 2012 à Paris par Vincent Pelletier de Nexedi. Quelques slides de ce document ont également été présentés lors du DZUG 2010 à Dresde.

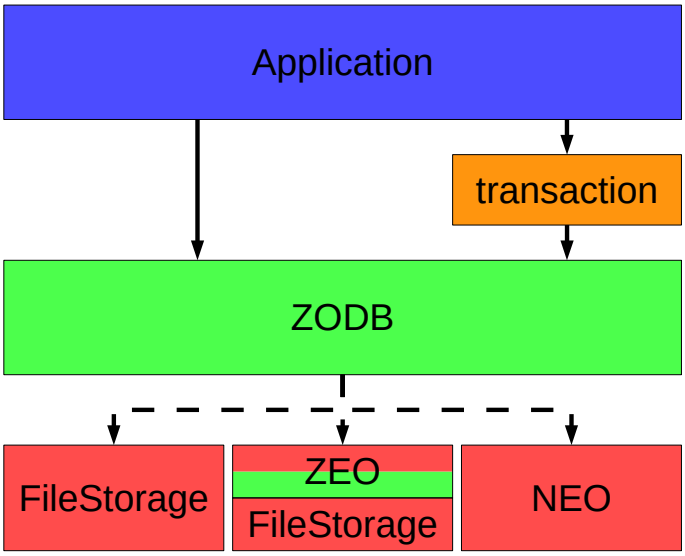
Une BDD objet: ZODB

NEO

- Répartition des données
- Transactions et parallélisme
- Reprise sur sinistre

Avant d'entrer dans les spécificités de NEO, voici un apperçu du fonctionnement de la ZODB (Zope Object DataBase).

Environnement



La ZODB est utilisée via deux interfaces: La première est directement accédée par l'application utilisant la base objet et permet le requêtage de la base. La seconde est accédée par le module de gestion des transaction, qui se charge des opérations de validation ou d'annulation de transaction lorsque l'application désire terminer une transaction.

En interne, la ZODB défini une API apellée ZODB.Storage, permettant des implémentations indépendantes du méchanisme de stockage des données. L'implémentation par défaut de cette interface est FileStorage, qui stocke toute la base dans un seul fichier sans gestion de partage d'accès.

ZEO (Zope Enterprise Object) gère ce partage d'accès and intercalant une couche réseau réseau entre la ZODB et le stockage. Il est techniquement possible d'utiliser toute implémentation ZODB.Storage derrnière ZEO, mais cela n'a d'avantages que pour celles n'implémentant pas leur propre couche réseau.

Il existe encore d'autres implémentations de l'API ZODB.Storage, stockant les données dans plusieurs fichiers ou dans des bases de données relationelles.

Fonctionnalités

Vue hiérarchique (niveau ZODB)

Vue objet (niveau ZODB.Storage)

La ZODB étant une base objet, l'unité de stockage est l'objet. Chaque objet est atteignable via un identifiant unique (OID, Object IDentifier), tout comme chaque ligne d'une base relationelle est atteignable via une clé primaire. La ZODB fourni une abstraction de cet identifiant via une notion de hiérarchie dans les objets stockés: l'OID d'un sous-objet est stocké dans son objet parent, et ce jusqu'à un objet racine avec un OID prédéfini.

Le scénario représenté correspond aux étapes (transactions) suivantes:

- T0: création d'un objet (ici, la racine de la base)

- T1: ajout de 4 sous-objets. On notera la modification de la racine, qui correspond à l'ajout des références des sous-objets.

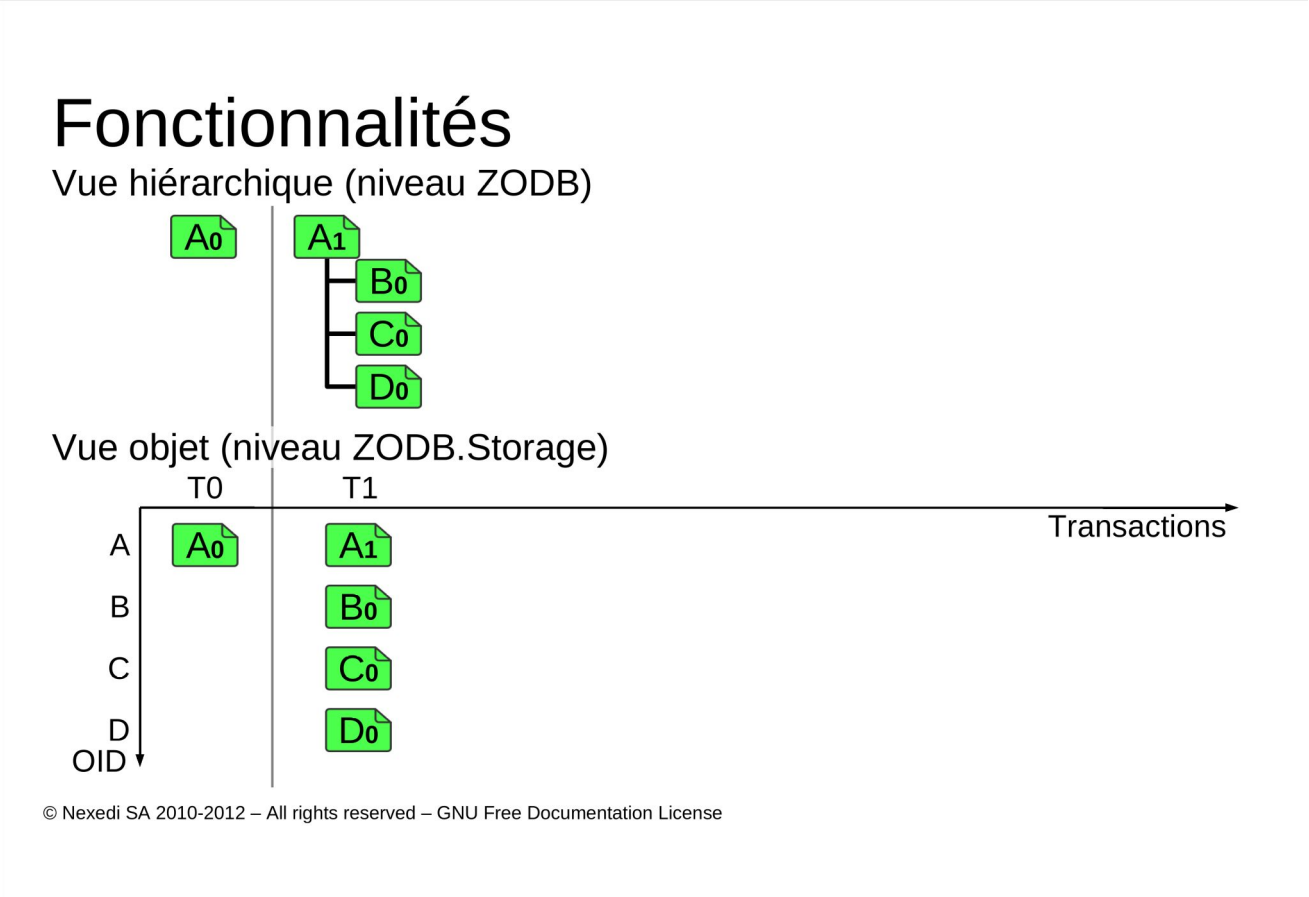
- T2: suppression d'un sous-objet (D). L'objet parent est de nouveau modifié pour la suppression de la référence vers D, mais ce dernier n'est pas modifié: il reste présent pour quiconque connait son identifiant.

- T3: modification de C. Il n'y a pas de modification de l'objet parent, puisque la référence de C ne change pas.

- T4: annulation de T4. Il n'y a pas stockage des données de C, puisqu'elles existent déjà dans la base. Un simple pointeur sera alors utilisé pour référencer la révision originale.

- T5: suppression de D. C'est en réalité tout simplement une application de l'annulation présentée à la transaction précédente, mais annulant la création d'un objet isolé: un pointeur nul est utilisé.

A tout moment, il reste possible de lister l'historique de chaque objet, et de charger les anciennes révisions. Il est possible de supprimer les anciennes révisions afin de récupérer l'espace qu'elles occupent.



La ZODB étant une base objet, l'unité de stockage est l'objet. Chaque objet est atteignable via un identifiant unique (OID, Object IDentifier), tout comme chaque ligne d'une base relationelle est atteignable via une clé primaire. La ZODB fourni une abstraction de cet identifiant via une notion de hiérarchie dans les objets stockés: l'OID d'un sous-objet est stocké dans son objet parent, et ce jusqu'à un objet racine avec un OID prédéfini.

Le scénario représenté correspond aux étapes (transactions) suivantes:

- T0: création d'un objet (ici, la racine de la base)

- T1: ajout de 4 sous-objets. On notera la modification de la racine, qui correspond à l'ajout des références des sous-objets.

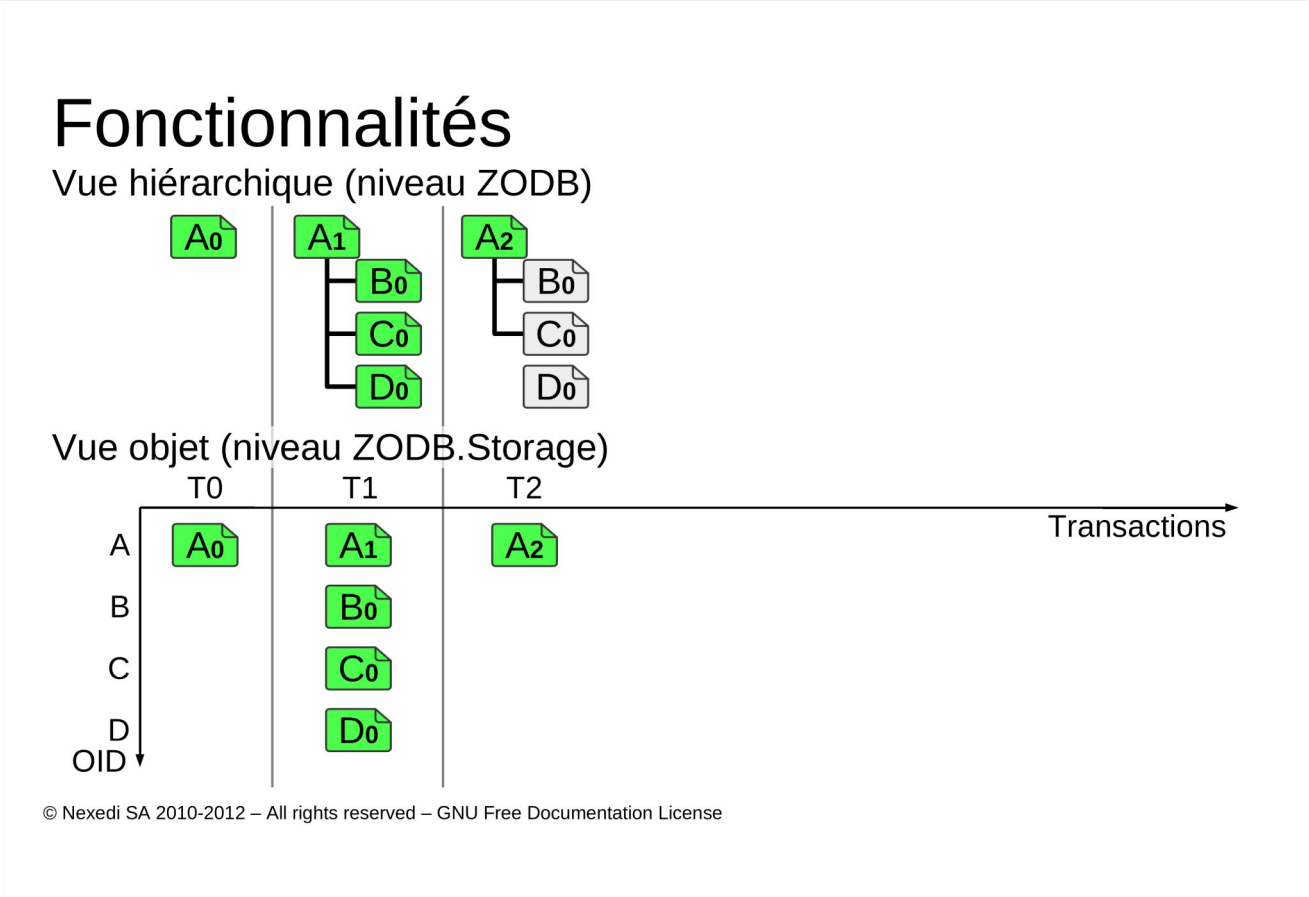
- T2: suppression d'un sous-objet (D). L'objet parent est de nouveau modifié pour la suppression de la référence vers D, mais ce dernier n'est pas modifié: il reste présent pour quiconque connait son identifiant.

- T3: modification de C. Il n'y a pas de modification de l'objet parent, puisque la référence de C ne change pas.

- T4: annulation de T4. Il n'y a pas stockage des données de C, puisqu'elles existent déjà dans la base. Un simple pointeur sera alors utilisé pour référencer la révision originale.

- T5: suppression de D. C'est en réalité tout simplement une application de l'annulation présentée à la transaction précédente, mais annulant la création d'un objet isolé: un pointeur nul est utilisé.

A tout moment, il reste possible de lister l'historique de chaque objet, et de charger les anciennes révisions. Il est possible de supprimer les anciennes révisions afin de récupérer l'espace qu'elles occupent.



La ZODB étant une base objet, l'unité de stockage est l'objet. Chaque objet est atteignable via un identifiant unique (OID, Object IDentifier), tout comme chaque ligne d'une base relationelle est atteignable via une clé primaire. La ZODB fourni une abstraction de cet identifiant via une notion de hiérarchie dans les objets stockés: l'OID d'un sous-objet est stocké dans son objet parent, et ce jusqu'à un objet racine avec un OID prédéfini.

Le scénario représenté correspond aux étapes (transactions) suivantes:

- T0: création d'un objet (ici, la racine de la base)

- T1: ajout de 4 sous-objets. On notera la modification de la racine, qui correspond à l'ajout des références des sous-objets.

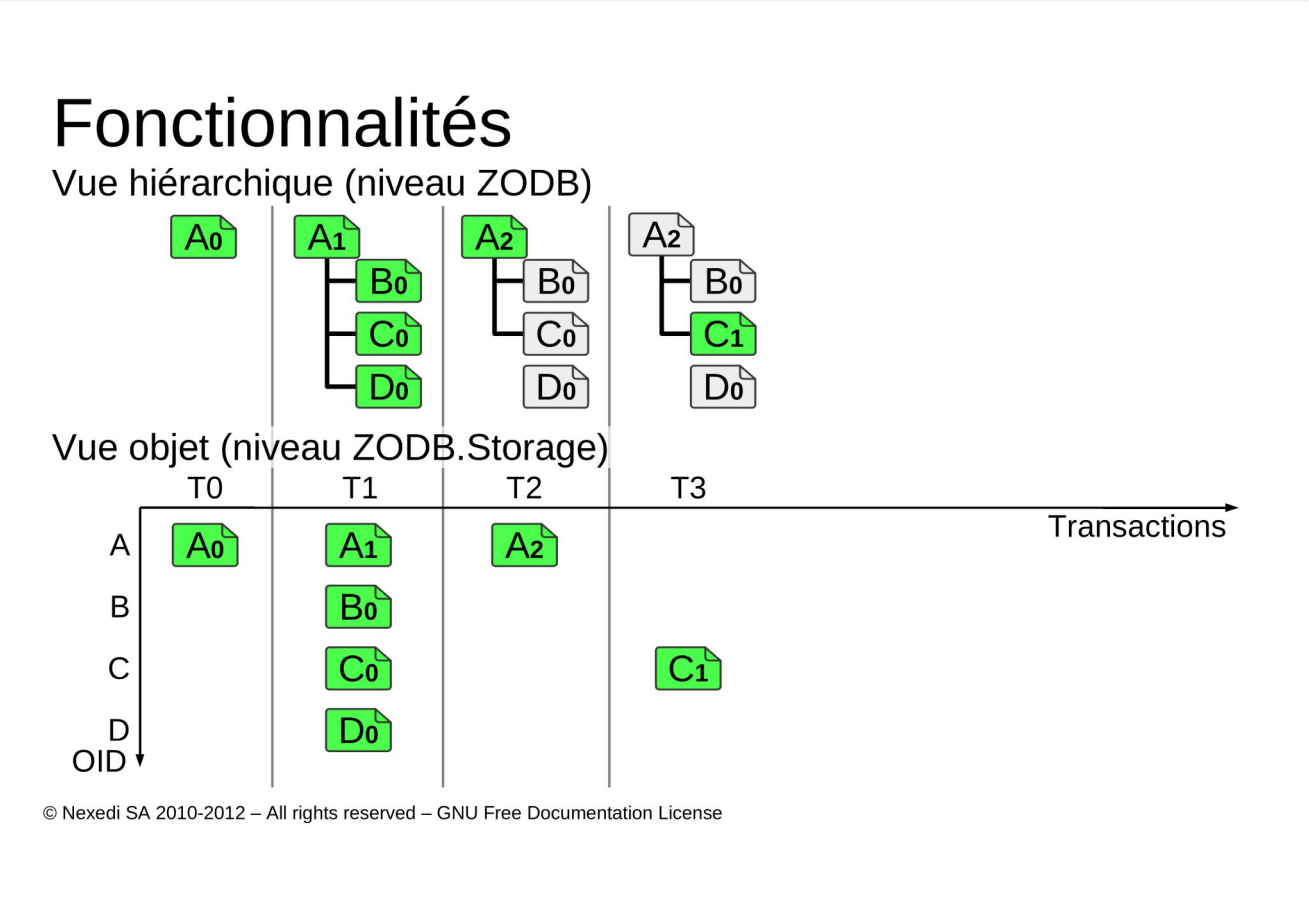
- T2: suppression d'un sous-objet (D). L'objet parent est de nouveau modifié pour la suppression de la référence vers D, mais ce dernier n'est pas modifié: il reste présent pour quiconque connait son identifiant.

- T3: modification de C. Il n'y a pas de modification de l'objet parent, puisque la référence de C ne change pas.

- T4: annulation de T4. Il n'y a pas stockage des données de C, puisqu'elles existent déjà dans la base. Un simple pointeur sera alors utilisé pour référencer la révision originale.

- T5: suppression de D. C'est en réalité tout simplement une application de l'annulation présentée à la transaction précédente, mais annulant la création d'un objet isolé: un pointeur nul est utilisé.

A tout moment, il reste possible de lister l'historique de chaque objet, et de charger les anciennes révisions. Il est possible de supprimer les anciennes révisions afin de récupérer l'espace qu'elles occupent.



La ZODB étant une base objet, l'unité de stockage est l'objet. Chaque objet est atteignable via un identifiant unique (OID, Object IDentifier), tout comme chaque ligne d'une base relationelle est atteignable via une clé primaire. La ZODB fourni une abstraction de cet identifiant via une notion de hiérarchie dans les objets stockés: l'OID d'un sous-objet est stocké dans son objet parent, et ce jusqu'à un objet racine avec un OID prédéfini.

Le scénario représenté correspond aux étapes (transactions) suivantes:

- T0: création d'un objet (ici, la racine de la base)

- T1: ajout de 4 sous-objets. On notera la modification de la racine, qui correspond à l'ajout des références des sous-objets.

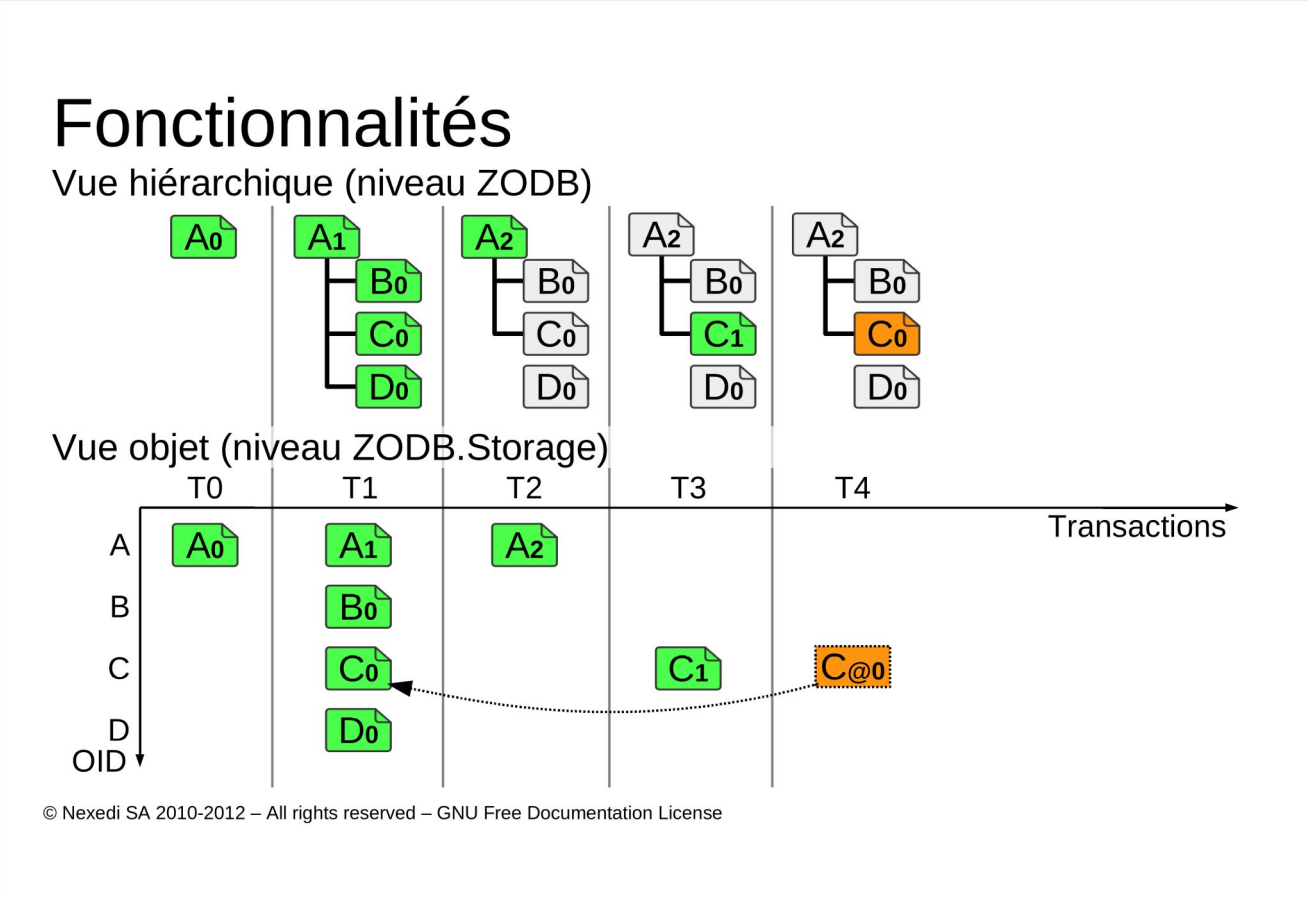
- T2: suppression d'un sous-objet (D). L'objet parent est de nouveau modifié pour la suppression de la référence vers D, mais ce dernier n'est pas modifié: il reste présent pour quiconque connait son identifiant.

- T3: modification de C. Il n'y a pas de modification de l'objet parent, puisque la référence de C ne change pas.

- T4: annulation de T4. Il n'y a pas stockage des données de C, puisqu'elles existent déjà dans la base. Un simple pointeur sera alors utilisé pour référencer la révision originale.

- T5: suppression de D. C'est en réalité tout simplement une application de l'annulation présentée à la transaction précédente, mais annulant la création d'un objet isolé: un pointeur nul est utilisé.

A tout moment, il reste possible de lister l'historique de chaque objet, et de charger les anciennes révisions. Il est possible de supprimer les anciennes révisions afin de récupérer l'espace qu'elles occupent.



La ZODB étant une base objet, l'unité de stockage est l'objet. Chaque objet est atteignable via un identifiant unique (OID, Object IDentifier), tout comme chaque ligne d'une base relationelle est atteignable via une clé primaire. La ZODB fourni une abstraction de cet identifiant via une notion de hiérarchie dans les objets stockés: l'OID d'un sous-objet est stocké dans son objet parent, et ce jusqu'à un objet racine avec un OID prédéfini.

Le scénario représenté correspond aux étapes (transactions) suivantes:

- T0: création d'un objet (ici, la racine de la base)

- T1: ajout de 4 sous-objets. On notera la modification de la racine, qui correspond à l'ajout des références des sous-objets.

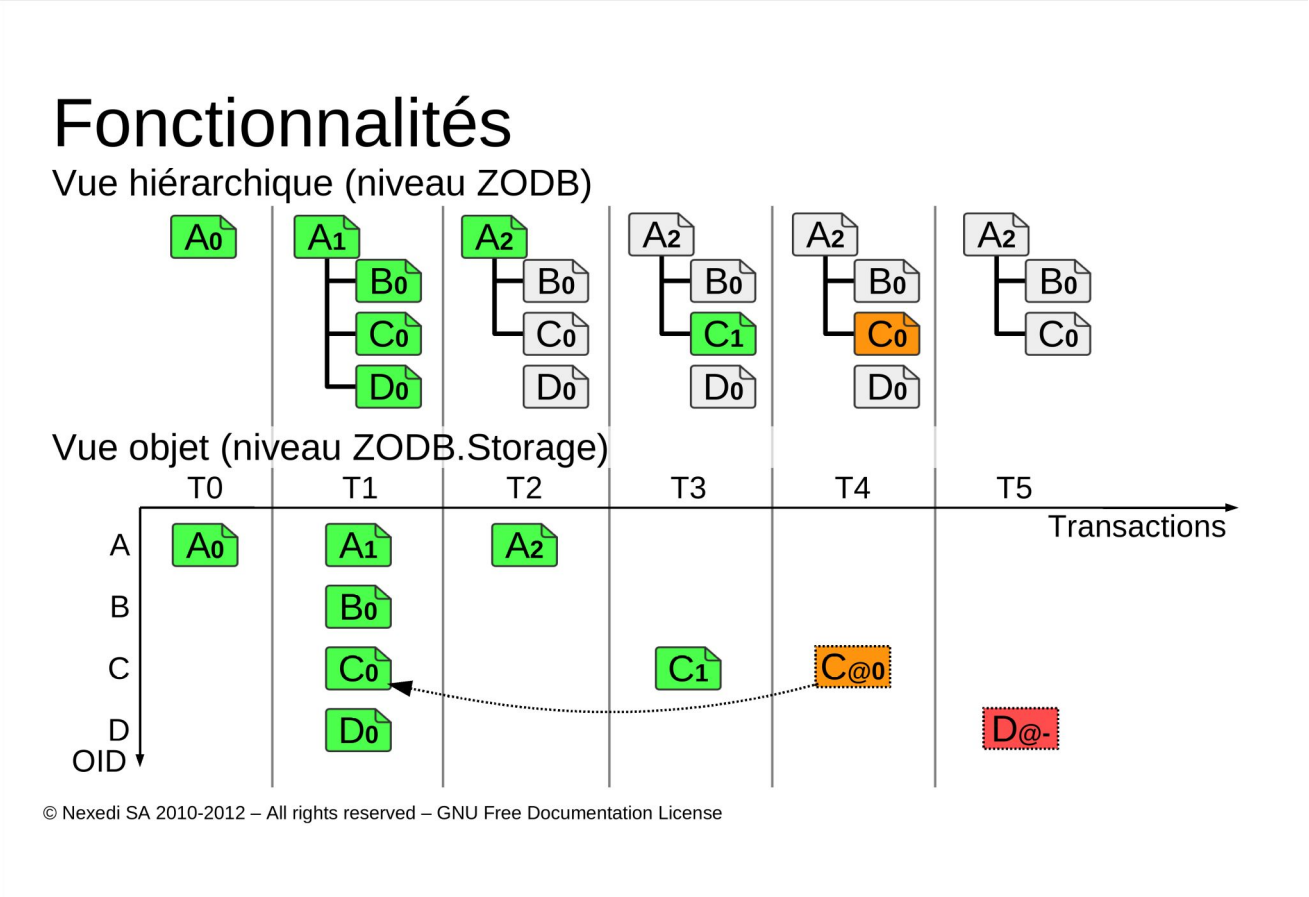
- T2: suppression d'un sous-objet (D). L'objet parent est de nouveau modifié pour la suppression de la référence vers D, mais ce dernier n'est pas modifié: il reste présent pour quiconque connait son identifiant.

- T3: modification de C. Il n'y a pas de modification de l'objet parent, puisque la référence de C ne change pas.

- T4: annulation de T4. Il n'y a pas stockage des données de C, puisqu'elles existent déjà dans la base. Un simple pointeur sera alors utilisé pour référencer la révision originale.

- T5: suppression de D. C'est en réalité tout simplement une application de l'annulation présentée à la transaction précédente, mais annulant la création d'un objet isolé: un pointeur nul est utilisé.

A tout moment, il reste possible de lister l'historique de chaque objet, et de charger les anciennes révisions. Il est possible de supprimer les anciennes révisions afin de récupérer l'espace qu'elles occupent.



La ZODB étant une base objet, l'unité de stockage est l'objet. Chaque objet est atteignable via un identifiant unique (OID, Object IDentifier), tout comme chaque ligne d'une base relationelle est atteignable via une clé primaire. La ZODB fourni une abstraction de cet identifiant via une notion de hiérarchie dans les objets stockés: l'OID d'un sous-objet est stocké dans son objet parent, et ce jusqu'à un objet racine avec un OID prédéfini.

Le scénario représenté correspond aux étapes (transactions) suivantes:

- T0: création d'un objet (ici, la racine de la base)

- T1: ajout de 4 sous-objets. On notera la modification de la racine, qui correspond à l'ajout des références des sous-objets.

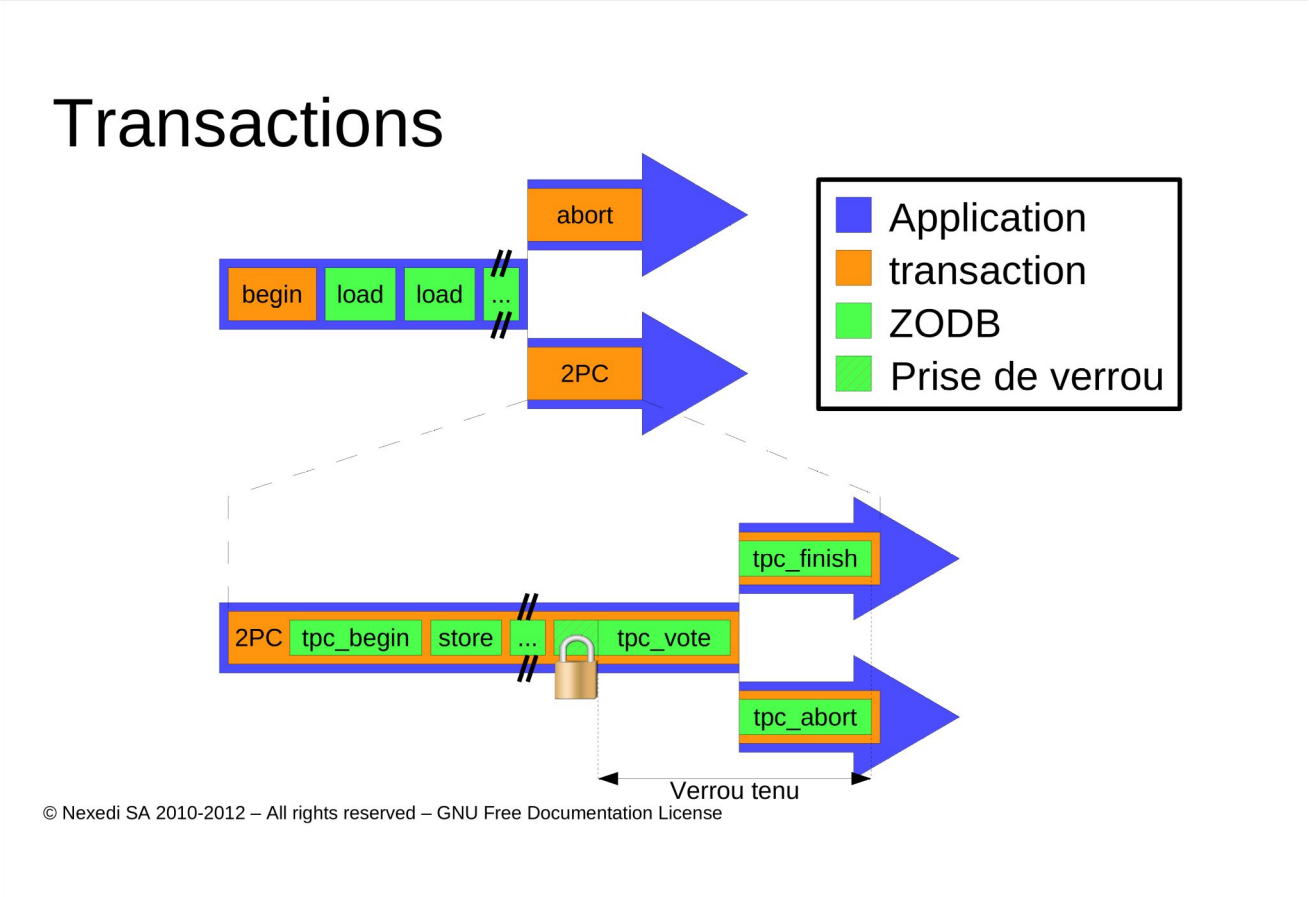
- T2: suppression d'un sous-objet (D). L'objet parent est de nouveau modifié pour la suppression de la référence vers D, mais ce dernier n'est pas modifié: il reste présent pour quiconque connait son identifiant.

- T3: modification de C. Il n'y a pas de modification de l'objet parent, puisque la référence de C ne change pas.

- T4: annulation de T4. Il n'y a pas stockage des données de C, puisqu'elles existent déjà dans la base. Un simple pointeur sera alors utilisé pour référencer la révision originale.

- T5: suppression de D. C'est en réalité tout simplement une application de l'annulation présentée à la transaction précédente, mais annulant la création d'un objet isolé: un pointeur nul est utilisé.

A tout moment, il reste possible de lister l'historique de chaque objet, et de charger les anciennes révisions. Il est possible de supprimer les anciennes révisions afin de récupérer l'espace qu'elles occupent.



La ZODB est transactionelle, comme impliqué précédement. Elle fourni un niveau d'isolation comparable à “Repeatable Read” (lectures répétables) utilisant un instantané de la base pris lors du premier accès de la transaction, suivant le méchanisme MVCC (MultiVersion Concurrency Control).

ZODB utilise une approche dite “optimiste” pour la détection de conflits de modification: la modification d'un objet ne pose aucun verrou sur le moment, mais s'appuie sur l'étape de validation de la transaction. Cette étape implémente le protocole de commit en deux phase: lors de la première phase, les données réellement modifiées par la transaction sont mises à disposition de la ZODB qui peut opposer un véto à la rétention du résultat de la transaction. Ce n'est qu'à cette étape qu'il est nécessaire, pour assurer la finalité de cette décision, qu'un ou plusieurs verrous soient pris. En cas d'avis positif, le commit final est effectué et les donées modifiées par la transaction deviennent atomiquement visibles à toute nouvelle transaction. Dans le cas contraire, la transaction est simplement annulée.

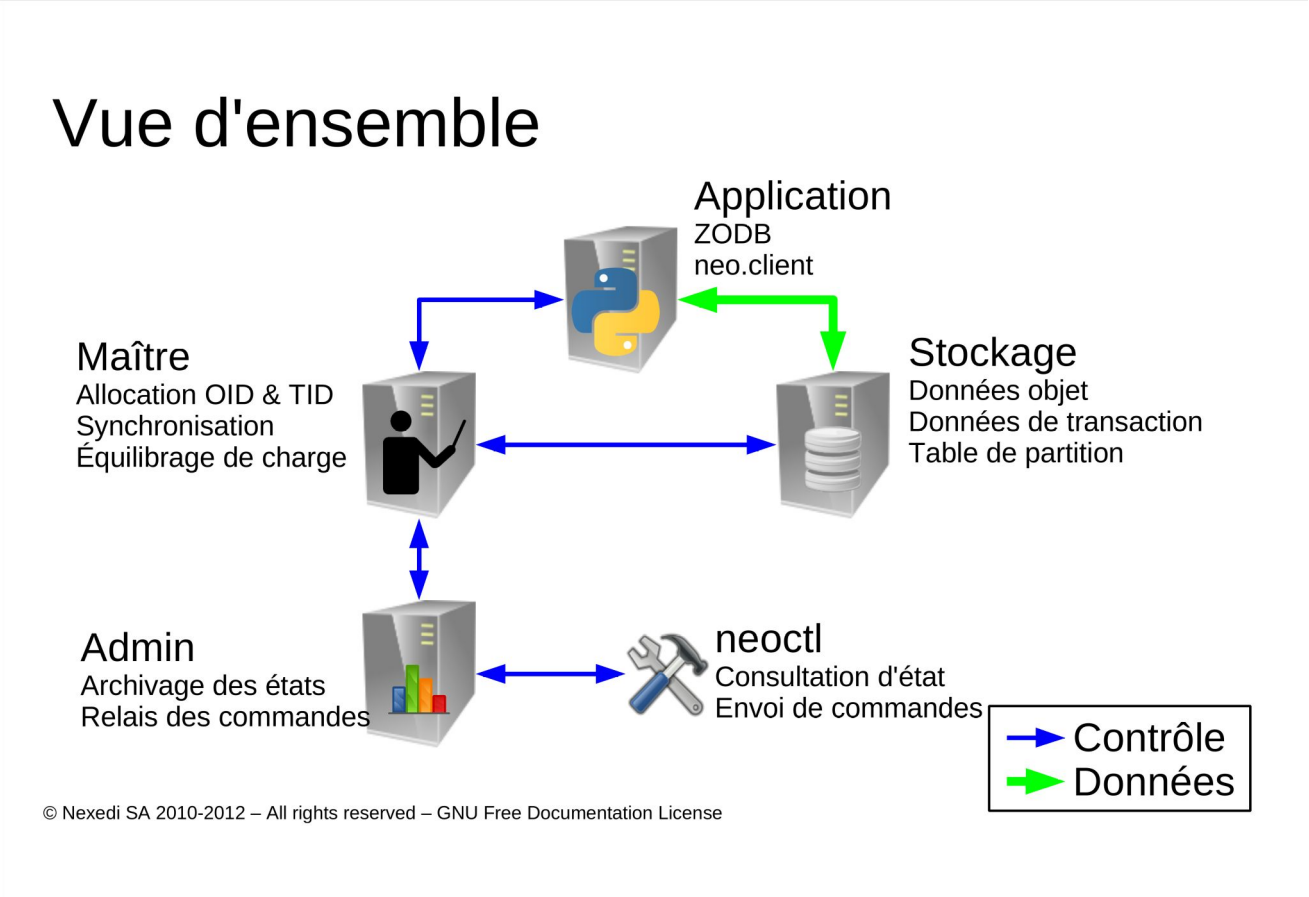
Lors de la détection d'un conflit de modification, la ZODB permet d'implémenter pour chaque type d'objet stocké un méchanisme de résolution de conflit afin de réduire la

La ZODB est particulièrement efficace lorsque l'application évite, de par son implémentation, de créer des points de contention – c'est à dire que les modifications sont régulièrement réparties dans la base, plutôt que concentrées sur un petit nombre d'objets. Elle est aussi facilement parallélisable, n'ayant qu'un seul point de contention réel qu'est l'étape de vote du commit en deux phases.

Une BDD objet: ZODB

NEO

- Répartition des données
- Transactions et parallélisme
- Reprise sur sinistre



Voici une vue d'ensemble des composants de NEO.

Au niveau de l'application se trouve la partie dite “client” de NEO, utilisée au travers de la ZODB.

Les noeuds de type maître sont des processus fournissant des fontionalités de générations de numéros séquentiels à fins d'utilisation en identifiants d'objets (OID, Object IDentifier) et de transactions (TID, Transaction IDentifier), le contrôle des transaction, et prend les décisions concernant l'équilibrage de charge des noeuds de stockage.

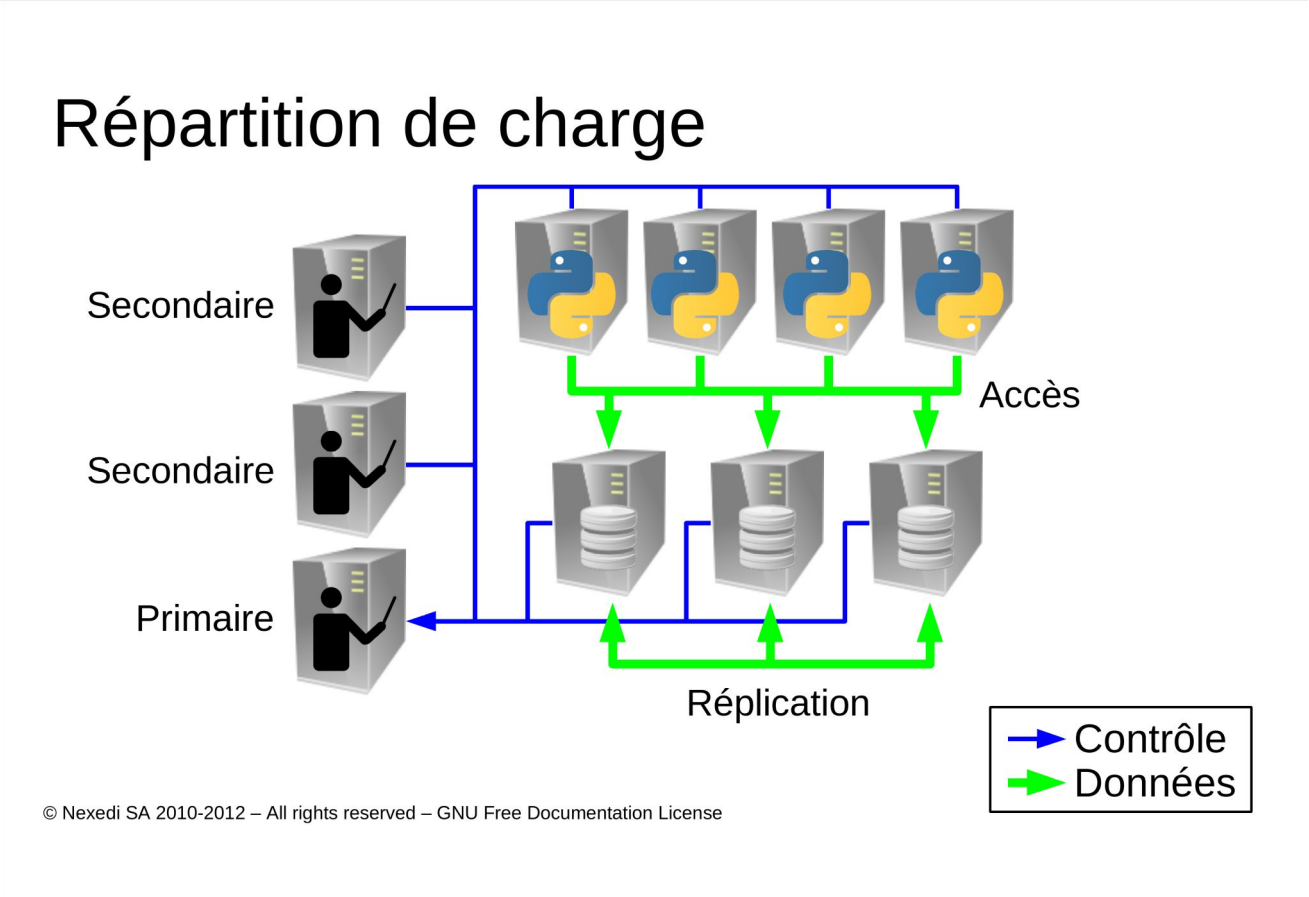
Les noeuds de stockage sont des processus fournissant l'accès à un média de stockage de données persistantes. Ils contiennent donc les données des objets et transactions qui composent la base objet totale, ainsi que quelques métadonnées nécessaires à NEO: table de partition et l'état des générateurs d'identifiants des noeuds maîtres.

Les noeuds d'administration sont à l'écoute des notification provenant d'un noeud maître, et les archive pour fournir un service de surveillance de la santé du cluster aux administrateurs humains. Il sert en outre de relai pour les commandes d'administration.

Neoctl est un outil en ligne de commandes permettant d'interroger et d'effectuer des actions sur un cluster, telles que l'ajout ou la suppression d'un noeud de stockage..

Toutes les interactions entre noeuds se font par réseau. Il est important de noter que les données des objets ne transitent que par les noeuds stockage et client, jamais par un noeud maître.

Une BDD objet: ZODB
NEO
Répartition des données
Transactions et parallélisme
Reprise sur sinistre



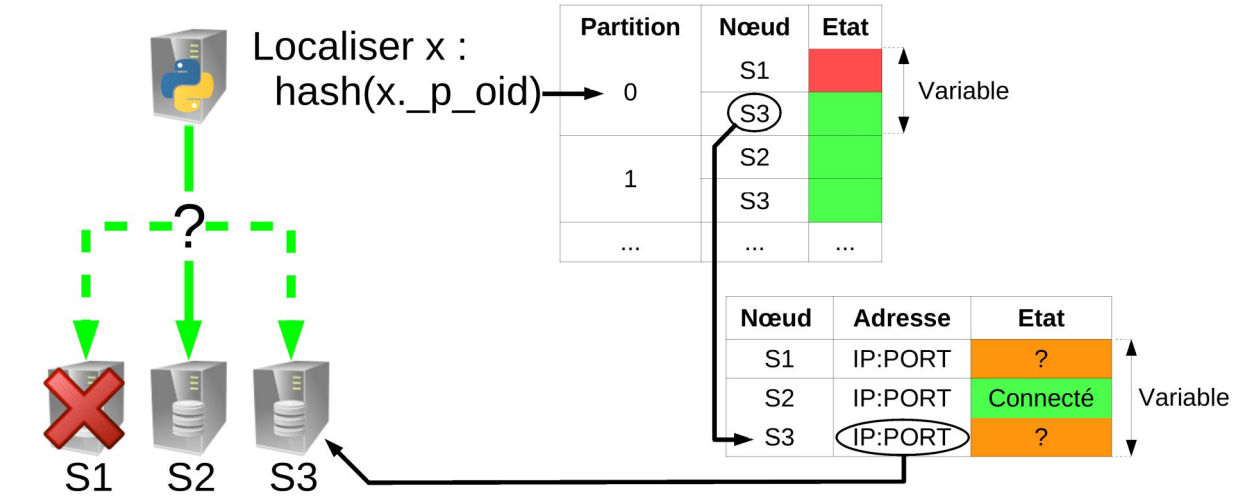
Afin de s'affranchir des limites matérielles de puissance de calcul (au niveau de l'application) et de bande passante des média de stockage, NEO permet de déployer chaque service en plusieurs exemplaires.

Tout d'abord, comme présenté précédement, NEO permet de partager l'accès la une même base. Il est donc possible de déployer plusieurs processus ou plusieurs machines faisant tourner une même application, accédant à des données communes.

NEO permet également de répartir une même base de données sur plusieurs médias de stockage, dans plusieurs machines: il fractionne la base de donnée pour la répartir entre plusieurs noeuds de stockage, et duplique ces portions selon le même principe que le RAID1+0: l'impact de la défaillance d'un certain nombre de média est limité par le nombre de réplicats, et la bande passante disponible pour l'application est augmentée par la quantité de fractionnement.

Dans le cas des noeuds maître, le fonctionnement est différent: les noeuds de type maître concentrant toutes les opérations que l'on ne peut pas distribuer dans un cluster NEO, un seul noeud maître est actif à un instant donné. Cependant, pour palier aux défaillance de ce noeud, il est néanmoins possible de mettre en place des noeuds supplémentaires qui seront prêts à prendre le relai sans intervention humaine.

Localiser un objet : table de partition



L'application a donc besoin de pouvoir localiser un noeud de stockage contenant ou pouvant recevoir un objet donné. La correspondance est obtenue en accédant à une table de partition et la liste de noeuds composant le cluster, connues de chacun des noeuds d'un cluster NEO et mises à jour par le maître primaire.

La table de partition fourni la correspondance entre un OID et un numéro de partition. On applique une fonction de hachage sur l'OID de l'objet considéré, ce qui donne la ligne de la table de partition à consulter. Sur cette ligne, on trouve plusieurs cellules, chacune correspondant à un noeud de stockage auquel cette ligne de partition est affectée. Le nombre de lignes de partition est fixe, mais chaque ligne de partition peut être affectée à un nombre variable de noeuds de stockage: cela correspond au nombre de réplicats souhaité pour la base de donnée, soit le nombre de noeuds qu'il est souhaité de pouvoir perdre sans rendre la base indisponible.

Chaque cellule de la table de partition contient un état permettant de savoir si le noeud de stockage contient des données à jour pour cette partition: ici, le noeud de stockage S1 est indisponible, et s'il contient toujours des données il est de toute façon certain qu'elles ne sont plus à jour. Il ne sera donc pas utilisé. Le noeud S3 est donc choisi, et recherché dans la liste des noeuds pour en connaître l'adresse afin de s'y connecter. Pour la même raison que pour les cellules de chaque ligne de la table de partition, la taille de la liste des noeuds est variable.

La pré-existance d'une connexion vers un noeud est également prise en compte dans le choix du noeuds de stockage à utiliser: si l'on avait cherché à charger un objet depuis la partition 43 dans l'exemple donné, on aurai alors préféré le noeud S2 à l'autre noeud candidat (S3).

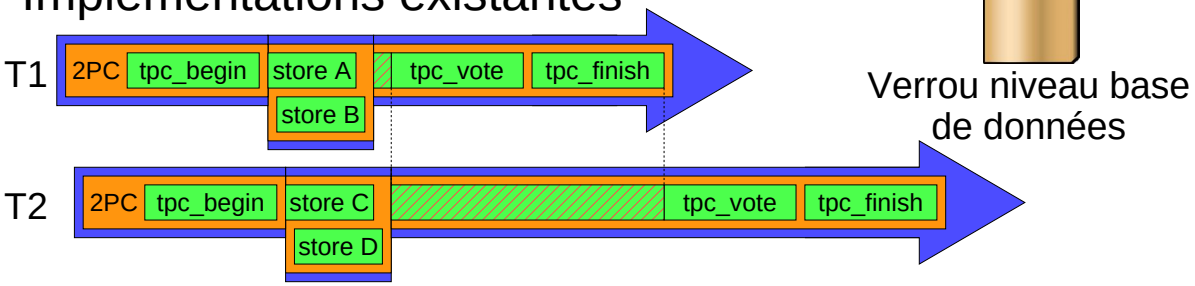
Une fois le noeud choisi, il peut être joint par son addresse, présente dans la liste des noeuds.

L'accès à une partition n'implique donc pas le noeud maître, ce qui limite les échanges réseau nécessaires et évite donc une augmentation de la latence.

Une BDD objet: ZODB
NEO
Répartition des données
Transactions et parallélisme
Reprise sur sinistre

Verrouillage

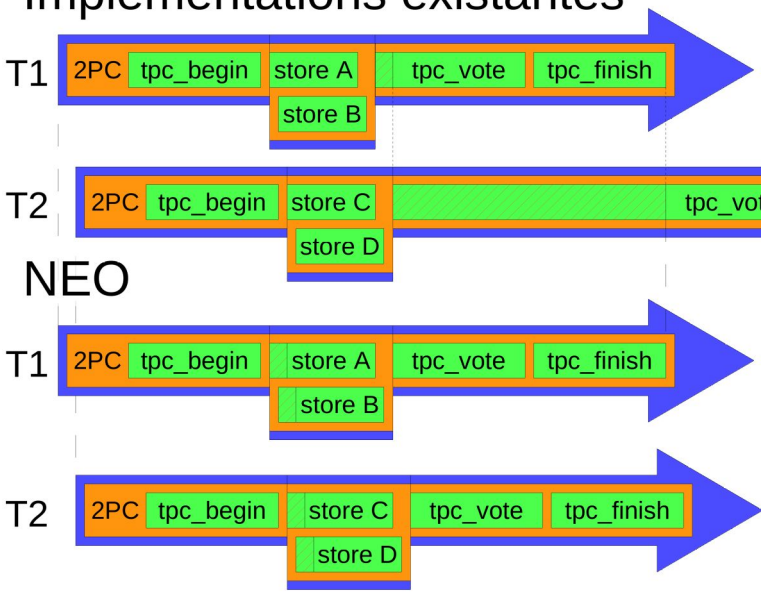
Implémentations existantes



Verrou niveau base de données

Verrouillage

Implémentations existantes



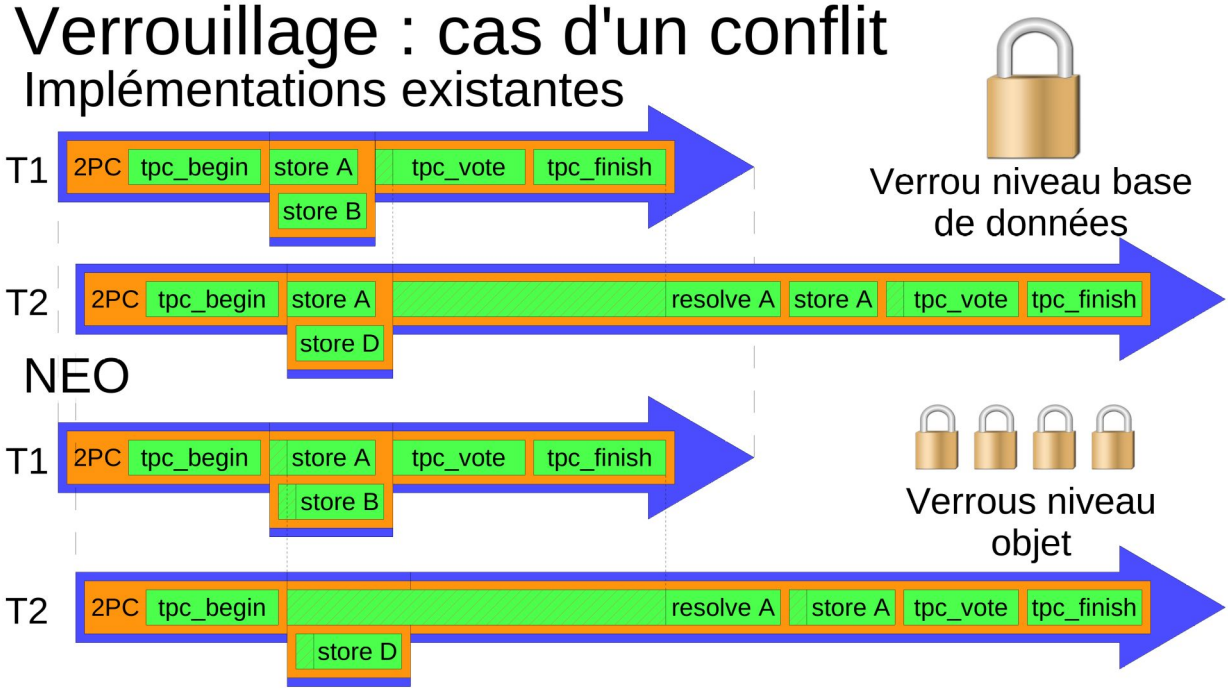
Verrou niveau base de données

NEO

Verrous niveau objet

Verrouillage : cas d'un conflit

Implémentations existantes



NEO

Une BDD objet: ZODB

NEO

- Répartition des données
- Transactions et parallélisme
- Reprise sur sinistre

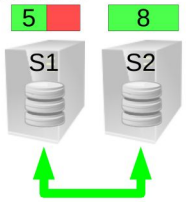
Déconnexion : réplication

1 Fonctionnement

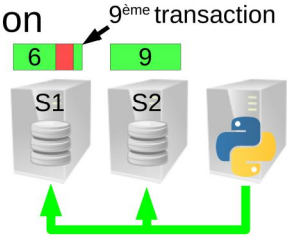
Partition	Nœud	Etat				
		1	2	3	4	5
0	S1					
	S2					
...						

2 Déconnexion

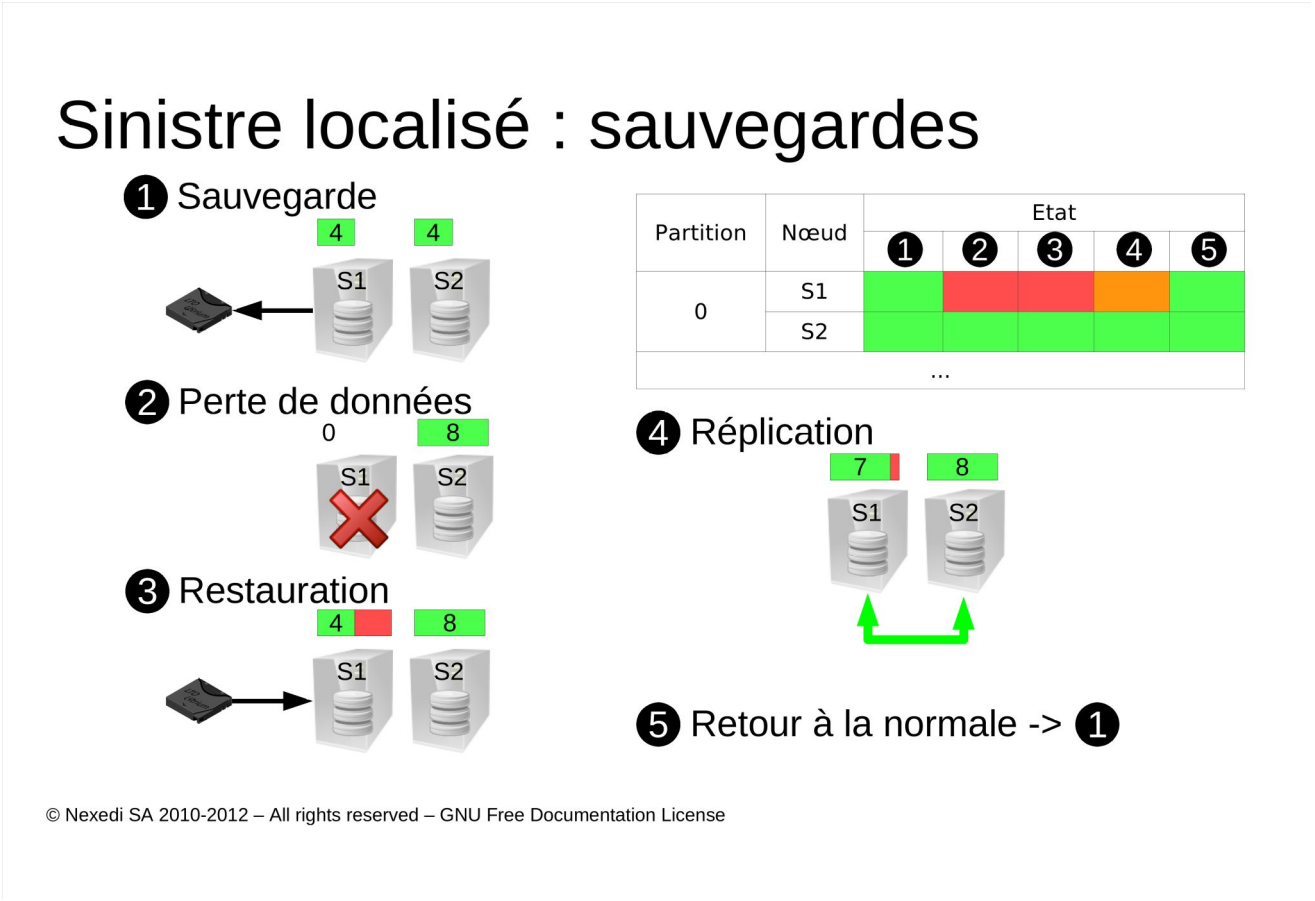
3 Reconnexion



4 Nouvelle transaction en cours de réplication



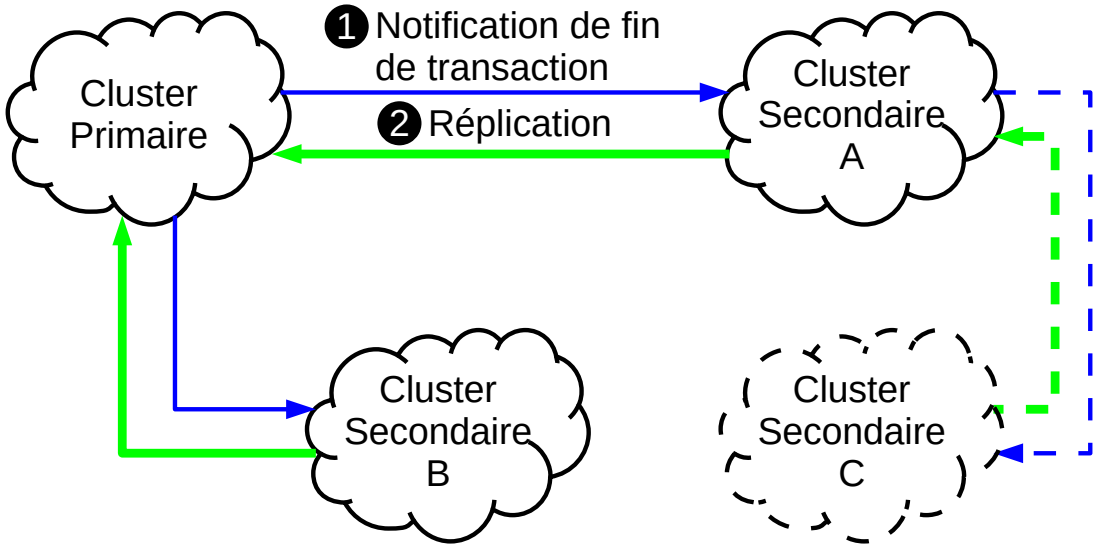
5 Retour à la normale -> 1



Chaque noeud de stockage peut gérer ses propres sauvegardes indépendemment et se remettre de problèmes impactant suffisament peu de noeuds pour que le service global soit préservé (c'est à dire, il reste au moins une copie de chaque objet de la base disponible durant le sinistre). Restaurer les données depuis la sauvegarde de chaque noeud impliqué dans la panne et le démarrage des services suffira pour déclencher la réplication des données depuis les noeuds survivants assignés aux mêmes partitions, et les noeuds nouvellement restaurés rattrapperont alors leur retard. C'est tout simplement le méchanisme normal de synchronisation de NEO, qui survient dès qu'un noeud se retrouve temporairement isolé du cluster.

De cette façon, le temps nécessaire pour restaurer les données n'impacte pas la disponibilité du cluster.

Sinistre étendu : réplication asynchrone



Lorsqu'un désastre de plus grande ampleur survient et impacte le fonctionnement global d'un cluster (perte de suffisament de données pour qu'une partie de la base ne soit plus disponible sur aucun noeud), le temps de restauration de backups entrerai alors en ligne de compte. Et s'il est moindre que le temps de restauration de l'intégralité de la base (puisque parallélisé entre chaque noeud de stockage), il reste proportionnel à la quantité de données à restaurer.

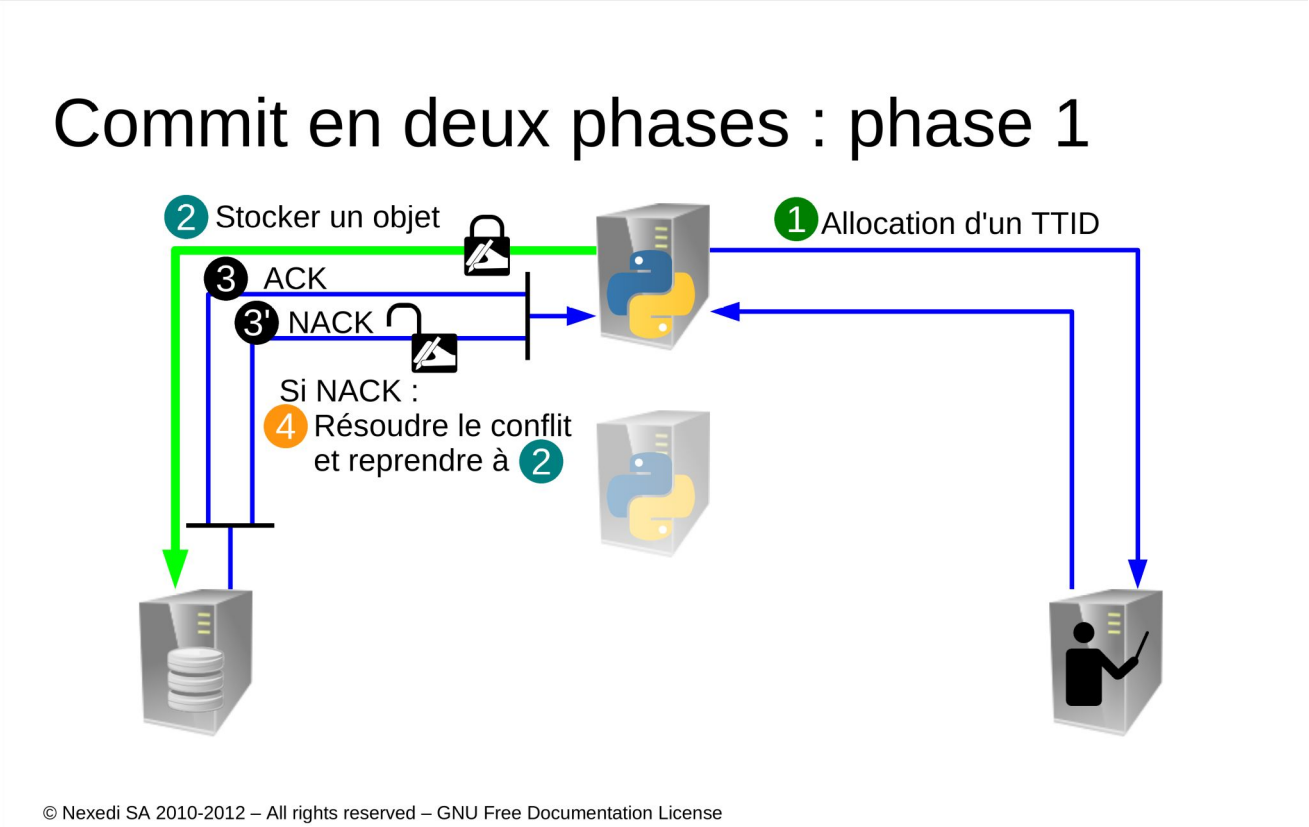
La durée d'une validation de transaction étant limitée par la latence entre un client et le noeud de stockage le plus lent, un déploiement de cluster NEO réparti entre plusieurs datacenters égraderai les performances de l'application.

La solution retenue dans ce cas de figure par NEO est de mettre en place un ou plusieurs clusters secondaires, physiquement éloigné du premier afin de réduire la probabilité de souffir des mêmes problèmes (coupure de courant, incendie, inondation, …) mais fonctionellement identiques au cluster principal (donc composés de noeuds maîtres, de stockage et éventuellement de copies de l'application inactives mais prêtes à prendre le relai). Dans cette configuration, les clusters secondaires reçoivent les notifications de fin de transaction du cluster principal. Ils y réagissent en déclenchant des réplications entre les noeuds de stockage qu'ils contiennent et les noeuds de stockage du cluster principal.

Lors d'une interruption de service du cluster principal, le cluster secondaire peut entrer très rapidement en service, moyennant la perte des opérations qu'il n'aurai pas eu le temps de répliquer.

Questions

Soutenez NEO !
www.neoppod.org



Le “commit en deux phases” débute par la demande d'un identifiant de transaction (TID, point 1) temporaire auprès du maître primaire (à moins qu'un TID spécifique n'ait été fourni par l'application, dans le cas d'un import depuis une base existante).

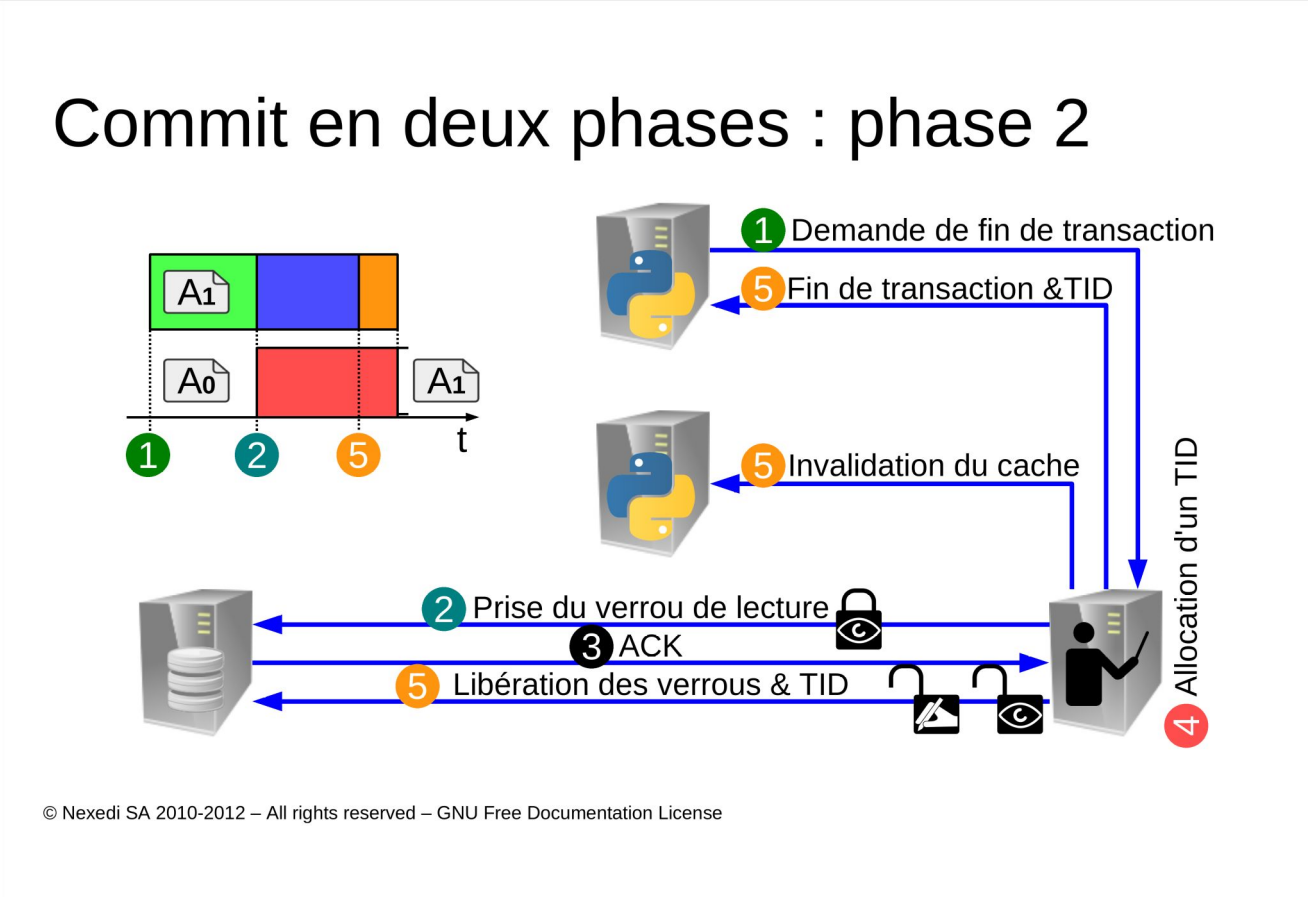
Ensuite, les données des objets sont envoyées aux noeuds de stockage (point 2).

Envoyer un objet a pour effet la prise d'un verrou d'écriture sur cet objet, qui est local à chaque noeud de stockage. Si ce verrou a déjà été pris lorsque le noeud de stockage recoit les données de l'objet, le client est mis en attente jusqu'à la libération du verrou.

Lorsque le verrou est accordé à un client, la dernière révision de l'objet dans la base est comparée à la version sur laquelle le client s'est basé (information qu'il transmet au noeud de stockage en même temps que les nouvelles données de l'objet): s'il y a une différence, le client est informé de la présence d'un conflit, et le verrou est libéré (point 3'). Sinon, le client est informé du succès de l'opération (point 3).

Une fois que le client a envoyé tous les objets, il passe à l'étape de vote pendant laquelle il attend toutes les réponses des noeuds de stockage. Si l'une de ces réponses signale une situation non-résolvable (conflit sur un objet n'implémentant pas de résolution de conflit), une erreur est levée à destination du méchanisme de gestion de transaction qui annulera alors la transaction. Cette annulation déclenche le relâchement de tous les verrous en écriture tenus.

Dans le cas contraire, la seconde phase de commit débute.



La seconde phase du commit débute lorsque le client demande au maître primaire de terminer la transaction (point 1), en lui envoyant la liste des objets impliqués dans la transaction. Le maître primaire prendra alors un verrou en lecture sur ces objets sur tous les noeuds de stockage concernés (point 2). Ce verrou de lecture permet de rendre les modifications atomiques pour les noeuds clients: une fois le verrou relâché, les effets de la transaction deviennent tous visible simultanément.

Lorsqu'un noeud de stockage reçoit la demande de verrouillage en lecture des objets d'une transaction, il écrit les données concernées dans une table temporaire, afin de garder une trace de la demande de finalisation de la transaction en cas de problème entre ce point et la fin de cette phase de commit (plantage ou indisponibilité d'un noeud).

Une fois que tous les noeuds de stockage ont confirmé la prise du verrou de lecture (point 3), le maître primaire alloue un identifiant définitif pour la transaction (point 4) et déclenche plusieurs actions (points 5).

Il demande à tous les noeuds de stockage concernés de relâcher ce verrou, ce qui a pour effet d'également relâcher les verrous d'écriture et de rendre la nouvelle version des objets disponible.

L'allocation tardive du TID permet d'éviter d'avoir à sérialiser globalement les transactions, tout en conservant la correspondance TID / instantané de l'état de la base, et permettant donc d'être utilisé comme référence MVCC pour l'isolation de transaction.

Egalement, le maître primaire informe les client n'ayant pas fait partie de la transaction de la présence de nouvelles révisions pour les objets impliqués dans la transaction, pour qu'ils les invalident dans leurs caches.

Simultanément, le maître primaine informe le client ayant déclenché la transaction de la fin de la seconde phase de commit: la transaction est maintenant stockée de façon permanente.