Active Data : Un modèle pour représenter et programmer le cycle de vie des données distribuées

Anthony Simonet

ComPAS'2014 : Parallélisme / Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Active Data : Un modèle pour représenter et programmer le cycle de vie des données distribuées

Anthony Simonet

Inria, ENS de Lyon, Université de Lyon, CNRS
anthony.simonet@inria.fr

Résumé
Alors que la science génère et traite des ensembles de données toujours plus grands et dynamiques, un nombre croissant de scientifiques doit faire face à des défis pour permettre leur exploitation. La gestion de données par les applications scientifiques de traitement intensif des données requière le support de cycles de vie très complexes, la coordination de nombreux sites, de la tolérance aux pannes et de passer à l'échelle sur des dizaines de sites avec plusieurs péta-octets de données. Dans cet article, nous proposons un modèle pour représenter formellement les cycles de vie des applications de traitement de données et un modèle de programmation pour y réagir dynamiquement. Nous discutons du prototype d'implémentation et présentons différents cas d'études d'applications qui démontrent la pertinence de notre approche.

Mots-clés : gestion de données distribuées

1. Introduction

La science actuelle traite de gigantesques ensembles de données. Les simulations à large échelle, les nouveaux instruments scientifiques et observatoires en génèrent tous d'importants volumes qu'il est nécessaire de transférer, préparer, distribuer et partager dans des communautés dispersées géographiquement, afin d'en extraire de la connaissance. Cette tendance devient prédominante dans des domaines comme la bio-informatique, la physique des hautes énergies (par exemple le Large Hadron Collider au CERN et l'expérience DØ au Fermilab), l'analyse d'images satellites ou encore l'exploitation de divers réseaux de capteurs (par exemple en sismologie, océanographie ou dans l'étude de la vie sauvage).
Pour les exploiter, les scientifiques appliquent, en plus de leur code métier, des opérations de gestion telles que du nettoyage, des transferts, du filtrage, de la réplication ou du partage. Nous appelons *cycle de vie des données* l'ensemble des étapes par lesquelles les données passent entre le moment où elles entrent dans un ensemble de systèmes et le moment où elles les quittent. Les données entrent dans un système lorsqu'elles sont acquises par un instrument ou dérivées d'autres données déjà présentes dans le système ; elles le quittent lorsqu'elles sont physiquement supprimées ou lorsqu'elles sont déplacées vers un stockage permanent, en dehors du système. Entre ces deux points dans le temps, les données passent par une série d'états qui codent l'avancement de leur traitement.
Pour programmer le cycle de vie, c'est-à-dire pour orchestrer l'exécution des tâches de calcul et des opérations de traitement, les utilisateurs usent de moyens allant de simples scripts spécifiques à leur besoin particulier, peu réutilisables, à des systèmes de workflows plus complexes.
Cependant, le volume qu'occupent les données en jeu rendent cette orchestration particulièrement difficile à programmer. La multiplication du nombre de machines et de disques impliqués fait mécaniquement augmenter le nombre de pannes et d'événements non désirés. La taille des données oblige à en distribuer le stockage et le traitement, ce qui force différents programmes sur différentes infrastructures à travailler ensemble, alors même que ces programmes n'ont pas été conçus pour collaborer. Compliquant encore plus leur collaboration, tous les systèmes de gestion de données n'identifient et ne nomment pas les données de la même manière ; il est donc difficile de faire le lien entre toutes les copies

d'une même donnée dans plusieurs systèmes. Enfin les ensembles de données sont de plus en plus souvent dynamiques : leur taille peut varier, et des sous-ensembles de leur contenu peuvent changer, ce qui nécessite des efforts particuliers en termes de gestion de ressources et d'optimisations.
De plus, la vue des systèmes de gestion de données est intrinsèquement limitée : un système de gestion de base de données (SGBD) ne peut par exemple pas *voir* ce qu'il se passe en dehors de ses bases de données. Ainsi ces systèmes qui participent au cycle de vie ont une vue parcellaire des événements qui se produisent et une possibilité limitée d'y réagir. Tout au contraire, l'utilisateur possède une place privilégiée ; lui seul possède la connaissance qui peut permettre aux systèmes de récupérer de situations ou d'événements qu'ils ne savent pas gérer, comme l'existence d'une copie de données que le système a perdu sur une panne matérielle, ou pour implémenter des optimisations fines, telles que du calcul incrémental ou du placement de données intelligent en fonction de leurs modes d'accès.
Ces applications sont aujourd'hui d'une importance capitale pour faciliter l'exploitation des données de toutes les sciences et améliorer l'utilisation des ressources de calcul et de stockage. Ces optimisations sont cependant extrêmement difficiles à mettre en œuvre car les systèmes de gestion de données sont pour la plupart des boîtes noires, et il n'existe pas d'interface unifiée qui permette de récupérer les quelques informations que certains fournissent.
Pour combler ce manque nous proposons *Active Data*, un modèle formel et graphique qui permet de représenter, pour un système donné, le cycle de vie interne qu'il créé. Ces modèles peuvent ensuite être composés afin de représenter le cycle de vie complet des données traversant différents systèmes, de leur création à leur suppression. Pour chaque donnée, Active Data permet au programmeur de déterminer un ensemble d'événements à surveiller (par exemple la création d'une donnée, la réplication, la fin d'un transfert, la perte d'une donnée etc.) et de programmer les opérations à effectuer lorsque ces événements surviennent. Ce modèle de programmation permet de développer un large éventail d'applications de gestion du cycle de vie des données, tels que l'automatisation du stockage hiérarchique, l'analyse à n'importe quelle étape du cycle de vie, la coordination entre les mécanismes d'acquisition des données et le stockage distant, les réseaux de distribution de contenu, la gestion de données incrémentales et bien d'autres.

Cet article présente les contributions d'Active Data pour la description, l'automatisation et l'optimisation du traitement du cycle de vie des données distribuées et dynamiques. Notre modèle permet de représenter l'état concurrent de données distribuées sur différentes infrastructures. Il offre une représentation graphique des cycles de vie qui permet de décrire les systèmes pour leurs utilisateurs et maintient un espace de noms unique qui fait le lien entre les différents identifiants d'une même donnée. En plus du modèle, nous présentons le modèle de programmation Active Data qui permet au programmeur d'exécuter du code distribué au plus près des données, à chaque étape, du début à la fin de leur cycle de vie, de manière simple, sans avoir à gérer le parallélisme.

Le reste de cet article motive l'approche d'Active Data à travers une étude de cas (section 2), introduit le modèle de cycle de vie d'Active Data (section 3), son implémentation et les systèmes actuellement supportés (section 4), présente deux exemples d'utilisation (section 5), discute les avantages et limites de l'approche proposée (section 6). Nous concluons avec une vue d'ensemble de l'état de l'art (section 7) et une réflexion sur les perspectives de ce travail.

2. Étude de cas d'un système de gestion du cycle de vie de données

Dans cette section, nous étudions le cas d'un système particulier qui gère le cycle de vie de données scientifiques. Il s'agit de l'expérience *Advanced Photon Source* (APS) qui a lieu au Laboratoire National d'Argonne aux États-Unis [22]. Dans l'APS, environs 100To de données sont générées chaque jour par un accélérateur de particules. Dès leur production, les données brutes sont pré-traitées localement afin de réduire leur taille et d'extraire des méta-données, puis de les enregistrer dans un catalogue de données Globus [1]. Les données pré-traitées sont ensuite transférées vers des sites partenaires pour y subir divers traitements et analyses. Les résultats de ces analyses, ainsi que des données de provenance associées sont également stockés dans le catalogue de données Globus afin d'y être partagés avec la communauté.

1. http ://www.globusonline.org/

Enfin certaines données sont transférées vers un stockage permanent.
Actuellement, le cycle de vie des données de l'expérience APS est géré à l'aide d'un ensemble de scripts. Les différentes étapes du cycle de vie sont soit initiées par l'utilisateur, soit à travers des interfaces spécifiques (par exemple SSH).
En plus des problèmes de fiabilité et de passage à l'échelle de la gestion de son cycle de vie, ce cas d'utilisation met en lumière plusieurs autres objectifs :

1. *Coordination entre différents centres de données :* Comme ici, le cycle de vie des données dans les applications scientifiques implique souvent plusieurs centres de données. Les données brutes peuvent souvent être générées sur un site, tandis que d'autres collaborent pour les traiter et les stocker.

   De plus, nombre d'applications scientifiques, et particulièrement dans le domaine des réseaux de capteurs, doivent gérer des cycles de vie qui nécessitent de la coordination entre plusieurs machines ou centres de données. Par exemple, ces applications n'ont souvent besoin que d'un sous-ensemble de taille déterminée des données acquises par les capteurs attachés aux nœuds du réseau. Éviter de traiter et de transférer toutes les données des capteurs permet de réduire la charge du système, d'économiser de l'énergie et d'utiliser plus efficacement ses ressources. Ceci nécessite cependant des techniques distribuées de limitation du débit d'acquisition (*data throttling*), coordonnées entre plusieurs sites/nœuds. De cette manière l'application peut choisir de ne retenir que p échantillons des capteurs pendant une période donnée, rejetant le reste des données avant tout traitement.

2. *Extensibilité des systèmes de gestion du cycle de vie :* Les cycles de vies des données scientifiques ne sont pas figés dans le temps. De plus, lorsque les données sont partagées, comme cela est le cas avec l'APS, le système qui gère le cycle de vie des données doit être très extensible de manière à prendre en compte tous les traitements que les scientifiques vont vouloir appliquer.

3. *Traitement incrémental :* De nombreuses applications scientifiques et commerciales traitent des flux continus de données. Il est le plus souvent superflu de refaire un traitement complet pour y incorporer de nouvelles données et mettre à jour les résultats. Les systèmes de gestion des cycles de vies des données doivent offrir une granularité suffisante aux tâches de traitement pour leur permettre de ne traiter que les données nouvellement acquises lors du dernier cycle, dès que cela est possible.

4. *Facilité d'intégration aux systèmes existants :* En plus d'être extensible, le système de gestion du cycle de vie doit avoir une empreinte minime sur les systèmes existants pour y être intégré et détecter ou recevoir leurs événements, et pour exécuter leurs opérations. Il doit offrir des performances capables de passer à l'échelle et un surcoût minimal sur les performances de ces systèmes.

Pour faire face à tous ces défis, nous présentons dans les deux prochaines sections le modèle de cycle de vie d'Active Data, qui permet de représenter de manière générique le cycle de vie de données dynamiques et distribuées, et son implémentation, qui permet de programmer des applications qui réagissent à ces cycles de vie.

3. Le modèle de cycle de vie

Active Data propose un modèle pour représenter les cycles de vie des données distribuées. Celui-ci s'inspire des Réseaux de Petri [14] qui sont couramment utilisés pour modéliser des systèmes complexes qui mettent en jeux de la concurrence et des partages de ressources. Ici les Réseaux de Petri nous permettent de représenter l'état d'une donnée partagée entre plusieurs systèmes en tenant compte de l'état individuel de chacun de ses réplicas. Ils permettent également de représenter que plusieurs réplicas d'une même donnée peuvent exister au même instant, chacun dans un état différent, et de définir des dépendances entre l'état des données et les opérations de gestion à appliquer.
Un Réseau de Petri est classiquement un 5-uplet $PN = (P, T, F, W, M_0)$ où :
– $P = \{p_1, p_2, \ldots, p_m\}$ est un ensemble fini de places représentées par des cercles ;
– $T = \{t_1, t_2, \ldots, t_n\}$ est un ensemble fini de transitions représentées par des rectangles ;
– $F \subseteq (P \times T) \cup (T \times P)$ est un ensemble d'arcs d'une place vers une transition, ou d'une transition vers une place ;

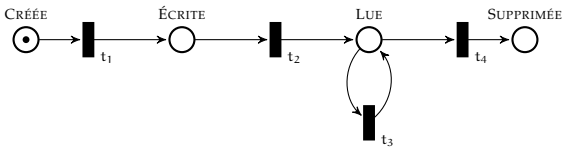


FIGURE 1 – Exemple de cycle de vie ou une donnée peut être écrite une seule fois, lue au moins une fois, puis supprimée.

– $W : F \to \mathbb{N}^+$ est une fonction de pondération qui indique combien de jetons chaque transition nécessite et combien de jetons elle produit.

Une place peut contenir zéro, un ou plusieurs jetons représentés par • . Une transition $t \in T$ est *activée* si et seulement si pour toute place p telle que $\{p \in P \mid (p,t) \in F\}$, le nombre de jetons dans p est supérieur ou égal à $w(p,t)$, le poids de l'arc entre p et t. Si ce poids est 1, il est généralement omit dans la représentation graphique.

Dans Active Data, un modèle de cycle de vie d'une donnée est un Réseau de Petri où les places représentent tous les états possibles de cette données, les transitions représentent tous les changements d'états possibles au cours du cycle de vie et où chaque jeton représente une copie de la donnée dans un état particulier. La figure 1 montre un exemple de cycle de vie *write-once/read-many*. Comme dans tous les cycles de vie, la donnée commence dans la place CRÉÉE. Cette donnée ne peut ensuite être écrite qu'une seule fois. Lorsque ceci se produit, la transition t_1 est déclenchée. La donnée doit ensuite être lue au moins une fois (transition t_2) et toute lecture supplémentaire est représentée par la transition t_3 qui fait une boucle sur la place LUE. Lorsque la donnée est supprimée, la transition t_4 est déclenchée et le jeton est déplacé sur la place SUPPRIMÉE.

En plus de la définition classique des Réseaux de Petri, chaque jeton porte deux identifiants ; l'un correspond à l'identifiant de la donnée dans le système qui la manipule et fait donc le lien avec le monde réel, l'autre permet de faire la différence entre toutes les copies d'une même donnée.

Enfin nous ajoutons des contraintes aux Réseaux de Petri pour garantir la cohérence des modèles de cycle de vie :

– Chaque modèle de cycle de vie doit commencer par une place CRÉÉE et terminer par une place SUPPRIMÉE ;
– Lorsque un jeton est présent dans la place SUPPRIMÉE, toutes les transitions sont désactivées.

La seconde contrainte est satisfaite à l'aide d'arcs inhibiteurs. La place SUPPRIMÉE représente donc la fin du cycle de vie, c'est-à-dire le dernier état qu'une donnée peut atteindre.

Un tel modèle représente le cycle de vie de données dans un système particulier. Pour permettre de représenter le cycle de vie de données de bout en bout, alors qu'elles traversent plusieurs systèmes, Active Data permet également de composer ces modèles entre eux. En particulier, Active Data propose un mécanisme pour faire le lien entre les identifiants des données dans tous les systèmes qui composent le cycle de vie, et ainsi offrir une vue transparente du cycle de vie à travers les systèmes en effaçant les détails des infrastructures.

Composer deux modèles de cycles de vie A et B implique de faire passer un jeton de A vers B en utilisant une *transition de composition*. Cette transition spéciale lie une place de A à la place CRÉÉE de B. Ainsi un jeton peut être consommé dans A alors qu'un autre est produit dans B. La transition de composition représente l'exécution du code nécessaire à la gestion des identifiants ; Active Data maintient un *catalogue de jetons* qui conserve, pour chaque donnée, les identifiants de toutes ses copies. Ceci permet, en connaissant une seule réplique d'une donnée, d'obtenir la liste de toutes les autres répliques et d'y accéder physiquement.

La figure 3 présente un exemple de composition entre un système de transfert de données (Globus Online) et un systèmes de stockage (iRODS). La transition t_{10} permet de représenter qu'une donnée (ici un fichier transféré) passe d'un système à un autre. La gestion des identifiants permet à chaque système d'examiner l'état de la donnée dans l'autre système. Nous présentons un exemple d'utilisation de cette composition dans la section 5.1.

4. Implémentation

Nous avons implémenté un prototype d'Active Data qui est composé de deux parties : i) un système d'exécution, qui gère le cycle de vie des données, garantit que l'exécution est correcte par rapport au modèle et transmet les transitions aux clients et ii) une interface de programmation (API) qui permet aux systèmes de gestion de données de déclarer les transitions qu'ils effectuent et aux programmeurs de développer des applications en enregistrant des fonctions de rappel (*callback*).
Cette section présente les fonctionnalités de l'interface de programmation et du système d'exécution, et présente une première évaluation de performance.

4.1. Modèle de programmation et système d'exécution

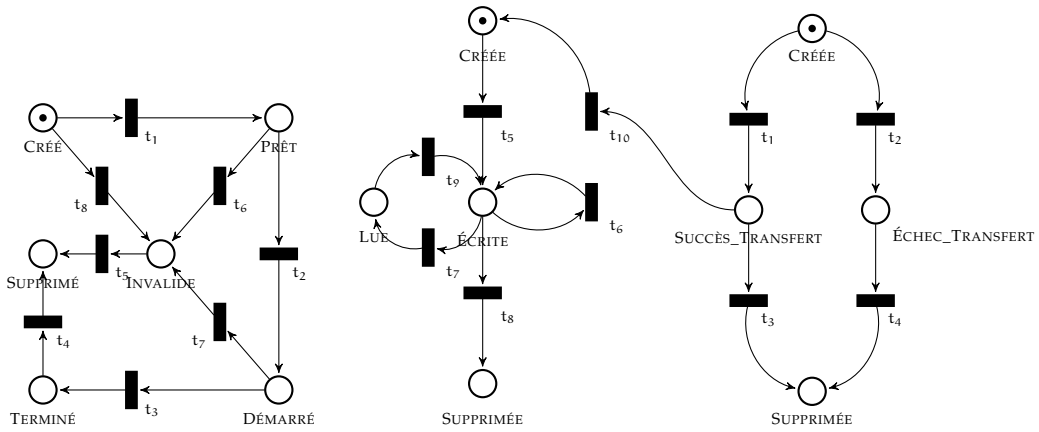
Active Data fonctionne ainsi : les systèmes de gestion de données exposent leur cycle de vie avec le formalisme défini dans la section 3. Toute action de leur part qui fait progresser une donnée dans son cycle de vie correspond à une transition dans le modèle et doit être signalée à un service centralisé appelé *Active Data Service* à l'aide de fonctions de l'API.
De leur côté, pour programmer des applications, les programmeurs fournissent des fonctions de rappel à l'API, afin de demander qu'elles soient exécutées à chaque fois qu'une transition est déclenchée.
Active Data utilise le paradigme Publish/Subscribe [7] pour propager jusqu'aux clients le fait qu'une transition a été déclenchée. Le Service Active Data reçoit toutes les transitions publiées par les systèmes de gestion de données et en informe les clients intéressés. Lorsqu'une notification est reçue par le client, le système d'exécution d'Active Data exécute localement les fonctions de rappel.
Pour assurer que les cycles de vie restent dans un état cohérent, le service stocke tous les cycles de vie et les met à jour lorsque des transitions sont publiées. De même, pour assurer la correction du système, les fonctions de rappel sont exécutées en série par les clients, dans l'ordre dans lequel les transitions ont été publiées ; ceci signifie que le service maintient un ordre total sur les transitions. De plus, les fonctions de rappel sont exécutées de manière bloquante par les clients mais asynchrones par rapport au service. Ainsi une fonction de rappel peut bloquer l'exécution des fonctions de rappel suivantes localement sur un client, mais elle ne peut pas bloquer le système entier. Afin d'éviter un blocage même local, ces fonctions doivent retourner rapidement, ou s'effectuer dans un thread séparé si une tâche particulièrement longue est programmée.
Active Data offre deux types de souscription pour enregistrer des fonctions de rappel : i) souscrire à une transition particulière pour n'importe quelle donnée qui la traverse ou bien ii) souscrire à une donnée particulière pour n'importe quelle transition qu'elle traverse. Pour éviter d'être submergé par des notifications, le service offre également une méthode qui retourne la copie d'un cycle de vie. De cette manière, les clients peuvent ponctuellement obtenir l'état d'un cycle de vie sans devoir récupérer toutes les transitions intermédiaires.
Une des fonctionnalités les plus intéressantes de l'implémentation du modèle est qu'elle permet au système d'exécution de contrôler dynamiquement la validité des transitions au moment où elles sont publiées, afin de vérifier qu'elles sont correctes vis-à-vis du modèle. Des exceptions sont levées dès qu'un client viole le modèle, c'est-à-dire lorsqu'un client essaie de : i) publier une transition qui n'appartient pas au modèle (opération illégale), ii) publier une transition qui n'est pas activée ou iii) publier une transition pour un cycle de vie qui est terminé.

4.2. Systèmes supportés

Nous avons actuellement modélisé le cycle de vie de quatre systèmes de gestion de données. BitDew [8] est un intergiciel développé à Inria qui permet de stocker et gérer facilement des données distribuées sur diverses infrastructures. BitDew permet de déclarer des propriétés en termes de placement, réplication et affinités sur les données et son système d'exécution les distribue en fonction. Le module inotify intégré au noyau Linux permet de surveiller un répertoire pour être notifié lors d'événements concernants ses fichiers, tels que la création, modification ou la suppression. iRODS [17], développé au centre DICE de l'Université de Caroline du Nord, est un système de stockage de données et de méta-données avec un catalogue de données virtuel qui offre un système de gestion par règles. Enfin Globus Online est un service de transfert de données développé au Laboratoire National d'Argonne qui permet de transférer rapidement et simplement de gigantesques volumes de données.

	Cycle de vie	Acquisition des événements	Accès au code source
BitDew	Complet	Instrumentation	Oui
inotify	Complet	Notifications	Oui
iRODS	Partiel	Déclencheurs PostgreSQL	Oui
Globus Online	Partiel	Notifications	Non

FIGURE 2 – Systèmes actuellement supportés par Active Data



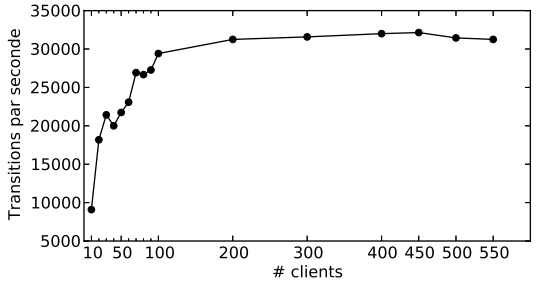
(a) Transferts de fichiers BitDew (b) iRODS (c) Globus Online

FIGURE 3 – Modèles de cycles de vie pour quatre systèmes de gestion de données.

Le prototype supporte de nombreux mécanismes pour collecter les événements sur les données. Par exemple, il est possible d'instrumenter le code source des systèmes auxquels nous avons accès, d'utiliser les systèmes de notification existants dans certains systèmes (inotify sur les systèmes de fichiers, déclencheurs dans les bases de données, etc.), de surveiller des journaux d'exécution (*logs*), ou même de capturer des notifications par courriel.
De plus, Active Data a une définition très flexible de ce qu'est une donnée, ce qui lui permet de s'adapter à la plupart des systèmes existants. Pour Active Data une donnée peut être soit un fichier, soit un n-uplet dans une base de données, soit tout objet dans un système, en fonction du besoin du programmeur.
Le tableau 2 résume pour chaque système comment son cycle de vie a été modélisé et comment les actions sur les données sont transformées en transitions puis publiées. Les modèles de cycles de vie sont présentés dans notre rapport technique [19].

4.3. Évaluation de performances

Nous présentons ici une première évaluation de performance afin de mesurer le débit, la latence et le surcoût d'Active Data sur les systèmes existants.
Les évaluations qui suivent ont été réalisées sur un cluster de la plateforme expérimentale Grid'5000 [3] composé de 92 nœuds, chacun équipé de 2 processeurs Intel Xeon L5420 quadri cœurs cadencés à 2,5Ghz. Chaque nœud est également équipé de 16Go de mémoire et d'un disque dur Sata II de 320Go. Les nœuds sont interconnectés par un réseau Ethernet Gigabit et utilisent Linux 2.6.32.
Nous évaluons le débit d'Active Data en mesurant le nombre de transitions que le système peut traiter par seconde. De manière à stresser le système, nous exécutons un serveur Active Data et un nombre variable de clients, chacun sur un nœud du cluster. Chaque client publie 10000 transitions dans une boucle, sans pause entre les itérations. La figure 4a présente le nombre moyen de transitions traitées chaque seconde par rapport au nombre de clients. Le service est ainsi capable de traiter jusqu'à 32000 transitions par seconde avant d'arriver à saturation.



(a) Nombre moyen de transitions par seconde traitées par le Service Active Data

		médiane	90ème centile	écart type
Latence	Local	0.77 ms	0.81 ms	18.68 ms
	Eth.	1.25 ms	1.45 ms	12.97 ms

		sans AD	avec AD
Surcoût	Eth.	38.04 s	40.6 s (4.6%)

(b) Latence et surcoût en millisecondes mesuré en utilisant BitDew avec et sans Active Data.

FIGURE 4 – Mesures de performances de l'implémentation prototype.

Nous évaluons la latence, c'est-à-dire le temps nécessaire pour informer le service de la création d'un nouveau cycle de vie et de l'exécution d'une transition et le surcoût d'Active Data, c'est-à-dire le temps supplémentaire dû à Active Data lors d'opérations de gestion de données. Nous prenons le système de transfert de fichiers de BitDew comme référence pour mesurer le surcoût. L'expérience consiste à transférer 1000 fichiers de 1Ko seulement et en même temps afin de surcharger le système. Lorsque Active Data est activé, plus de 6000 transitions sont publiées durant les transferts de fichiers. Le temps d'exécution est enregistré avec et sans Active Data.
Le tableau 4b montre la médiane, le 90ème centile et l'écart type pour la latence en millisecondes, lorsque le service et le client sont hébergés sur le même nœud (local), et lorsque les deux sont sur deux machines différentes (ether). Le surcoût est donné en seconde et en pourcentage comparé au BitDew de base. Nous observons que la latence est de l'ordre de la milliseconde, avec une stabilité remarquable. Le surcoût, même lorsque le système est stressé, reste inférieur à 5%.
Dans l'ensemble, ces expériences montrent que notre prototype est suffisamment performant pour offrir une certaine réactivité aux applications et passer à l'échelle lors de nos études de cas.

5. Exemples

Pour faire la démonstration du modèle de programmation Active Data, nous avons développé quatre systèmes avec des cycles de vie présentant des propriétés différentes : i) un cache pour Amazon S3, ii) un cycle de vie complexe pour effectuer de la coordination et de la réduction du débit d'acquisition distribuée dans un réseau de capteurs, iii) un environnement MapReduce incrémental, et iv) une solution pour capturer la provenance de données réparties dans plusieurs systèmes. Ces cas d'utilisation démontrent la capacité d'Active Data à simplifier le développement de cycles de vie complexes, les traitements incrémentaux, la collaboration entre plusieurs systèmes et à améliorer leur efficacité.
Dans la suite de cette section nous présentons ces deux dernières applications en détail.

5.1. Provenance de données multi-systèmes

La provenance des données constitue l'historique complet des opérations et dérivations au cours d'un cycle de vie. Elle est essentielle pour préserver la qualité des données scientifiques dans le temps. Maintenir la provenance de données de bout en bout est difficile car elle nécessite de pouvoir examiner l'ensemble des opérations effectuées au cours de leur cycle de vie, alors que même les systèmes spécialisés sont limités par leur *champs de vision*. Nous faisons ici la démonstration de notre solution qui reconstruit la provenance de données de systèmes hétérogènes en tirant parti de la vue unifiée du cycle de vie des données offerte par Active Data.
Nous étudions un scénario dans lequel des transferts de fichiers sont lancés depuis un dépôt Globus Online distant vers un espace de stockage local temporaire à quelques secondes d'intervalle. Dans ce scénario, Active Data reçoit des événements concernant des données qui évoluent indépendamment dans deux systèmes en parallèle : Globus Online, un service de transfert de fichiers et iRODS, un logiciel de stockage de données qui fournit également un catalogue de méta-données.

La figure 3 montre comment les deux systèmes sont composés pour représenter le passage d'une donnée d'un système vers l'autre.
Dans notre scénario, pour chaque fichier présent dans iRODS, nous voulons enregistrer les informations de provenance à propos de son transfert : quelles étaient la source et la destination du transfert, quand il a commencé et terminé et les éventuels échecs.
Active Data est l'élément qui permet à Globus Online et iRODS de voir la partie du cycle de vie des fichiers qui est en dehors de leur champ. Nous utilisons le catalogue de méta-données utilisateurs d'iRODS pour stocker les informations de provenance conjointement avec les fichiers de données.
Quand un transfert Globus Online démarre, une tâche de transfert est créée et son identifiant `Task Id` devient l'identifiant du jeton dans le modèle.
Pour composer les cycles de vie, la place SUCCÈS_TRANSFERT de Globus Online créée un jeton dans le modèle de cycle de vie d'iRODS. La réception du courriel notifiant le succès du transfert provoque le déclenchement de la transition t_1 et l'exécution d'une fonction de rappel qui stocke le fichier dans iRODS. iRODS retourne un identifiant `DATA_ID` qui se trouve à son tour ajouté au jeton ; celui-ci contient maintenant les deux identifiants.
Une seconde fonction de rappel est attachée à t_{10}, la transition de création d'iRODS : elle est exécutée lorsqu'une donnée est créée dans iRODS. Cette fonction de rappel demande le cycle de vie au service d'Active Data pour voir s'il contient un identifiant Globus Online. Le cas échéant, elle envoie une requête à Globus Online pour obtenir des informations sur le transfert qui a eu lieu.
Afin de démontrer notre solution, des transferts sont démarrés d'un dépôt Globus Online vers un stockage local à quelques secondes d'intervalle. Nous observons que lorsqu'un transfert termine, les méta-données à propos de celui-ci apparaissent immédiatement dans le catalogue de données d'iRODS avec le `Task Id` correct et d'autres méta-données (points de départ et d'arrivée, date de début et de fin du transfert). Le Listing 1 montre les méta-données accessibles pour un de ces fichiers présent dans iRODS après la fin d'un transfert.

```
$ imeta ls -d test/out_test_4628
AVUs defined for dataObj test/out_test_4628:
attribute: GO_FAULTS
value: 0
----
attribute: GO_COMPLETION_TIME
value: 2013-03-21 19:28:41Z
----
attribute: GO_REQUEST_TIME
value: 2013-03-21 19:28:17Z
----
attribute: GO_TASK_ID
value: 7b9e02c4-925d-11e2-97ce-123139404f2e
----
attribute: GO_SOURCE
value: go#ep1/~/test
----
attribute: GO_DESTINATION
value: asimonet#fraise/~/out_test_4628
```

Listing 1 – Méta-données associées à un fichier dans iRODS transféré avec Globus Online

La vue unique qu'Active Data offre des jeux de données distribués sur des systèmes hétérogènes permet ici la reconstruction de la provenance des données de bout en bout.

5.2. MapReduce incrémental

Dans cet exemple, nous montrons comment un système existant peut bénéficier d'Active Data pour gérer efficacement des données dynamiques.
Une des principales limites de MapReduce est son inefficacité face à des données mutables ; lorsque un job MapReduce est effectué plusieurs fois alors que seul un sous-ensemble des données en entrée ne change entre deux exécutions, toutes les tâches map et reduce doivent s'effectuer à nouveau. Rendre MapReduce incrémental, c'est-à-dire re-exécuter les tâches map et reduce seulement pour les données qui ont changé, nécessite de modifier profondément son flux de données complexe. Cependant, si l'environnement MapReduce est capable d'observer le cycle de vie des données qu'il manipule, il peut adapter dynamiquement ses calculs aux modifications des données.
Nous utilisons l'implémentation de MapReduce implémentée à partir de BitDew [20] et dont le modèle

Fraction modifiée	20%	40%	60%	80%
Temps de mise à jour	27%	49%	71%	94%

TABLE 1 – MapReduce incrémental : temps de mise à jour du résultat en fonction de la fraction des données d'entrée modifiée.

de cycle de vie est présenté dans la figure 3a. Dans cette implémentation, un nœud maître place les données d'entrées fragmentées dans une instance de BitDew et lance un job MapReduce dont les tâches map et reduce sont effectuées respectivement par des *mappeurs* et *réducteurs*. Cependant les fragment d'entrée peuvent être modifiés directement dans le système de stockage par des applications tierces. Pour rendre cette implémentation de MapReduce incrémentale, nous ajoutons simplement un drapeau "modifié" aux fragments d'entrée. Quand un fragment est marqué comme modifié, le mappeur qui l'avait précédemment traité exécute à nouveau la tâche map sur la nouvelle version du fragment et envoie une nouvelle donnée intermédiaire aux réducteurs. Sinon le mappeur retourne la donnée intermédiaire qu'il avait précédemment mémorisé. Les réducteurs calculent le résultat final comme à leur habitude. Pour mettre à jour le drapeau "modifié", nous devons faire réagir le maître et les mappeurs à des transitions dans le cycle de vie des fragments. Plus précisément, les nœuds souscrivent à deux transitions déclenchées par le système de stockage, grâce à Active Data (figure 3a) :

– t_3 est observée par le maître. Lorsque cette transition est déclenchée, le nœud maître vérifie que le transfert est local et qu'il modifie un fragment en entrée. Ceci se produit lorsque le maître place tous les fragments dans le système de stockage avant de lancer le job. Dans ce cas, une fonction de rappel marque le fragment comme modifié.
– t_1 est observée par les mappeurs. Quand cette transition est déclenchée, les mappeurs vérifient si le transfert est distant et si un de leur fragment d'entrée est modifié. Dans ce cas, la fonction de rappel du mappeur marque le fragment comme modifié.

Pour évaluer les performances de notre MapReduce incrémental, nous comparons le temps nécessaire au traitement du jeux de données complet avec le temps nécessaire à la mise à jour du résultat après avoir modifié une partie du jeux de données. L'expérience se déroule comme suit ; nous exécutons l'application *wordcount* avec 10 mappeurs et 5 réducteurs et un jeux de données de 3,2Go découpé en 200 fragments. Le tableau 1 présente le temps de mise à jour du résultat par rapport au temps de calcul complet lorsqu'une fraction variable du jeux de données est modifiée. Il demeure cependant un surcoût dû au fait que les phase de *shuffle* et *reduce* sont entièrement exécutées dans notre implémentation. De plus, les fragments modifiés ne sont pas distribués uniformément parmi les nœuds, ce qui provoque un déséquilibre de la charge. Des optimisations plus poussées pourraient sans doute diminuer ce surcoût mais nécessiteraient de modifier l'implémentation en profondeur. Cependant, nous montrons que nous obtenons un gain de performances significatif en modifiant moins de 2% du code source de l'environnement MapReduce.

Cet exemple montre comment la connaissance supplémentaire apportée par Active Data permet à des systèmes de s'adapter pour optimiser le traitement de données particulièrement dynamiques.

6. Discussion

6.1. Avantages

L'approche d'Active Data centrée sur les données apporte de nombreux avantages par rapport à une approche classique qui se concentre sur les tâches tels que les *workflows*.

– *Programabilité* : Active Data expose le cycle de vie des données d'une manière simple et graphique, ce qui permet au programmeur d'identifier rapidement les événements qui déclenchent l'exécution d'opérations et de les programmer.
– *Vérification* : Modéliser les cycles de vie de données permet de vérifier leur correction. De nombreux outils (par exemple CPNTools [2]) permettent d'automatiser la vérification de certaines propriétés sur les Réseaux de Petri, telles que l'absence d'interblocage, l'existence d'une borne ou la vivacité. Ils peuvent être utilisés pour vérifier ces propriétés sur des modèles de cycles de vie.

2. Colored Petri Net Tools (CPN Tools) : `http://cpntools.org/`

– *Facilité à déboguer* : La structure événementielle des programmes développés avec Active Data les rend faciles à déboguer. Le modèle sépare clairement les étapes de traitement des interactions entre ces étapes, ce qui rend les bogues rapides à identifier et à corriger.
– *Facilité à modéliser et implémenter la tolérance aux pannes* : Implémenter de la tolérance aux pannes dans les programmes développés avec Active Data est simple car les fautes et leur récupération peuvent être incluses dans le modèle de cycle de vie. Ainsi une faute est une transition, et le code pour la corriger s'implémente simplement dans une fonction de rappel.
– *Passage à l'échelle* : La vue orientée sur les événements d'Active Data permet de passer facilement à l'échelle, car elle distribue implicitement les tâches de gestion de données, sans composant centralisé pour les réaliser.
– *Facilité de distribution et d'optimisation* : L'exécution distribuée du code est implicite avec Active Data : les sites et nœuds qui contiennent les données exécutent les opérations déclenchées par Active Data ce qui, de plus, favorise la localité avec les données.
– *Possibilité de découpler gestion et opérations* : Le code client d'Active Data est extrêmement léger et peut être déployé sur des machines avec une très faible puissance de calcul et peu de mémoire, tandis que les opérations de gestion de données les plus lourdes peuvent être effectuées sur des architectures très puissantes, flexibles et fiables, telles qu'un nuage.
– *Interaction très fine avec le cycle de vie* : Contrairement à la programmation centrée sur les tâches où les opérations sur les données se passent typiquement au début ou à la fin des tâches avec des actions de pré- ou post-traitement, Active Data permet d'agir à chaque étape intermédiaire d'un processus ou à l'occasion d'événements particuliers. Par exemple un service de transfert de fichiers compatible avec Active Data rendra également possible l'exécution de code pendant le transfert, ou lorsque celui-ci est bloqué, afin de remédier à la situation.

6.2. Limites

– *Complexité* : D'une manière générale, en l'absence d'outils appropriés, il est plus difficile de raisonner sur des interactions complexes avec un système événementiel. Cette complexité peut-être partiellement maitrisée avec des outils qui permettent de modéliser et vérifier les cycles de vie, et de faciliter la programmation et le débogage des fonctions de rappel associées.
– *Absence de standard* : Dans un monde idéal, la majorité des systèmes qui manipulent des données fourniraient une interface Active Data. Ce n'est cependant pas le cas et les développeurs de systèmes ont la responsabilité de fournir une interface pour interagir avec les systèmes de gestion de données. Ce processus peut être simplifié en fournissant une méthodologie commune pour instrumenter le code existant, ou collecter des informations afin de reconstituer le cycle de vie des données. Active Data simplifie sensiblement cette tâche en supportant un large éventail de méthodes pour collecter des informations sur les événements qui altèrent les données (voir section 4).

7. État de l'Art

À notre connaissance, il n'existe pas d'autre système permettant de gérer le cycle de vie des données dans les applications scientifiques. Cependant, les travaux suivants sont pertinents par rapport à cet article.

Active Message.

Active Data emprunte à Active Message [21] l'idée d'exécuter du code fournit par l'utilisateur quand un message arrive. Dans la même direction, Active Disks [1] permet l'exécution de code directement au niveau du matériel de stockage pour alléger la charge processeur des machines de calcul et améliorer la localité des accès aux données.

Modèles de Programmation.

Un certain nombre de modèles de programmation ont émergé pour le traitement de données à grande échelle, et assez peu ont été largement adoptés. MapReduce [5] permet d'exprimer des programmes en fonction de deux primitives — map et reduce, Dryad [11] permet la programmation de flux de données parallèles, AllPairs [13] qui permet de faire des comparaisons paire-à-paire sur de grands jeux de

données, Swift [23] qui permet de programmer et automatiser l'exécution de grands flux de données. Il existe également des évolutions de ces paradigmes tels que PigLatin [15] qui fournit une interface de requête haut niveau à MapReduce ou Twister [6], un système d'exécution qui permet d'effectuer des calculs MapReduce de manière itérative. Quelques systèmes supportent des modes de calcul incrémentaux ou permettent de traiter des flux continus de données. Parmi eux nous pouvons citer Percolator [16], Nephele [12], MapReduce-Online [4]. Chimera [9] est quand à lui offre un moyen de représenter les dérivations de données et un système de requêtes. Bien que ces systèmes visent à fournir des modèles de programmation pour l'analyse de données dynamiques à grande échelle, ils ciblent seulement un système et n'aide aucunement à gérer le cycle de vie de données mettant en jeux plusieurs systèmes.

Systèmes de stockage.

Quelques systèmes de stockage permettent d'effectuer des optimisations au niveau des fichiers, par exemple optimiser le placement de fichiers, le niveau de réplication ou des caches par rapport à la manière dont les fichiers sont accédés. BitDew [8] permet aux utilisateurs de spécifier des comportements aux données en termes de tolérance aux pannes, réplication, protocole de transfert ou affinités dans le placement. MosaStore [2] est un système de stockage de fichier optimisé pour la plupart des schémas d'accès collectifs que l'on peut trouver dans les workflows (gather/scatter, reduce, broadcast). Active Data peut être intégré à BitDew, MosaStore et d'autres systèmes pour offrir des optimisations du stockage au niveau des fichiers.

8. Conclusion et Perspectives

Ce travail propose le cycle de vie des données comme nouveau paradigme pour modéliser les besoins en termes de données des applications scientifiques. Nous pensons que les applications scientifiques requièrent un paradigme fondamentalement nouveau pour gérer le cycle de vie des données, avec deux caractéristiques clefs : être *centré sur les données* (par opposition à centré sur les tâches) et être *événementiel* (par opposition à séquentiel). Nous présentons le modèle de programmation Active Data et un système qui implémente ce paradigme.
Les travaux futurs vont se concentrer sur plusieurs aspects. Le modèle peut être étendu dans les directions suivantes : une représentations plus fine des calculs, ce qui creuserait l'idée de consommation et production de données par les transitions ; la représentation de collections de données, qui permettraient des traitements collectifs sur les jeux de données. Concernant l'implémentation d'Active Data, nous prévoyons de travailler sur des mécanismes de rollback pour offrir des exécutions tolérantes aux fautes et évaluer des implémentations distribuées de la couche Publish/Subscribe. Enfin plusieurs prototypes d'application sont développés avec Active Data : un réseau de distribution de contenu distribué et coopératif qui permet la distribution d'images embarquant de lourdes applications de Physique des Hautes Énergies sur des grilles de PC [10] et un service distribué de stockage d'images de checkpoint avec une sélection du serveur en fonction de distances réseaux [18]. Nous travaillons également sur l'implémentation d'un système capable d'automatiser la gestion et le partage des données de l'expérience APS présentée plus haut.

Bibliographie

1. Acharya (A.), Uysal (M.) et Saltz (J.). – Active disks : Programming model, algorithms and evaluation. – In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 81–91, New York, NY, USA, 1998. ACM.
2. Al-Kiswany (S.), Gharaibeh (A.) et Ripeanu (M.). – The case for a versatile storage system. *ACM SIGOPS Operating Systems Review*, vol. 44, n1, 2010, pp. 10–14.
3. Bolze (R.) et all. – Grid'5000 : A large scale and highly reconfigurable experimental grid testbed. *Int. J. High Perform. Comput. Appl.*, vol. 20, November 2006, pp. 481–494.
4. Condie (T.), Conway (N.), Alvaro (P.), Hellerstein (J. M.), Elmeleegy (K.) et Sears (R.). – Mapreduce online. – In *Proceedings of the Seventh USENIX Symposium on Networked Systems Design and Implementation, NSDI'10*, volume 10, p. 20, San Jose, California, USA, 2010.
5. Dean (J.) et Ghemawat (S.). – Mapreduce : simplified data processing on large clusters. *Communications of the ACM*, vol. 51, n1, 2008, pp. 107–113.

6. Ekanayake (J.), Li (H.), Zhang (B.), Gunarathne (T.), Bae (S.-H.), Qiu (J.) et Fox (G.). – Twister : A runtime for iterative mapreduce. – In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, *HPDC '10*, HPDC '10, pp. 810–818, Chicago, Illinois, 2010. ACM.
7. Eugster (P. T.), Felber (P. A.), Guerraoui (R.) et Kermarrec (A.-M.). – The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, vol. 35, n2, 2003, pp. 114–131.
8. Fedak (G.), He (H.) et Cappello (F.). – Bitdew : a programmable environment for large-scale data management and distribution. – In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–12. IEEE, 2008.
9. Foster (I.), Vockler (J.), Wilde (M.) et Zhao (Y.). – Chimera : A virtual data system for representing, querying, and automating data derivation. – In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pp. 37–46. IEEE, 2002.
10. He (H.), Fedak (G.), Kacsuk (P.), Farkas (Z.), Balaton (Z.), Lodygensky (O.), Urbah (E.), Caillat (G.), Araujo (F.) et Emmen (A.). – Extending the egee grid with xtremweb-hep desktop grids. – In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 685–690. IEEE Computer Society, 2010.
11. Isard (M.), Budiu (M.), Yu (Y.), Birrell (A.) et Fetterly (D.). – Dryad : distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, vol. 41, n3, 2007, pp. 59–72.
12. Lohrmann (B.), Warneke (D.) et Kao (O.). – Massively-parallel stream processing under qos constraints with nephele. – In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, *HPDC '12*, HPDC '12, pp. 271–282, Delft, The Netherlands, 2012. ACM.
13. Moretti (C.), Bulosan (J.), Thain (D.) et Flynn (P. J.). – All-pairs : An abstraction for data-intensive cloud computing. – In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–11. IEEE, 2008.
14. Murata (T.). – Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, vol. 77, n4, Apr 1989, pp. 541–580.
15. Olston (C.), Reed (B.), Srivastava (U.), Kumar (R.) et Tomkins (A.). – Pig latin : a not-so-foreign language for data processing. – In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1099–1110. ACM, 2008.
16. Peng (D.) et Dabek (F.). – Large-scale incremental processing using distributed transactions and notifications. – In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, *OSDI'10*, volume 10, pp. 1–15, Vancouver, BC, Canada, 2010.
17. Rajasekar (A.), Moore (R.), Hou (C.-y.), Lee (C. A.), Marciano (R.), de Torcy (A.), Wan (M.), Schroeder (W.), Chen (S.-Y.), Gilbert (L.) et al. – irods primer : integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, vol. 2, n1, 2010, pp. 1–143.
18. Ratnasamy (S.), Handley (M.), Karp (R.) et Shenker (S.). – Topologically-aware overlay construction and server selection. – In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, *INFOCOM 2002*, volume 3, pp. 1190–1199. IEEE, 2002.
19. Simonet (A.), Fedak (G.) et Ripeanu (M.). – Active data : A programming model for managing big data life cycle. *Inria Technical Report* N°*RR-8062*, 2012.
20. Tang (B.), Moca (M.), Chevalier (S.), He (H.) et Fedak (G.). – Towards MapReduce for Desktop Grid Computing. – In *Fifth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'10)*, pp. 193–200, Fukuoka, Japan, November 2010. IEEE.
21. von Eicken (T.), Culler (D. E.), Goldstein (S. C.) et Schauser (K. E.). – Active messages : a mechanism for integrating communication and computation. – In *25 years of the international symposia on Computer architecture (selected papers)*, pp. 430–440. ACM, 1998.
22. Wang (Y.), De Carlo (F.), Mancini (D. C.), McNulty (I.), Tieman (B.), Bresnahan (J.), Foster (I.), Insley (J.), Lane (P.), von Laszewski (G.) et al. – A high-throughput x-ray microtomography system at the advanced photon source. *Review of Scientific Instruments*, vol. 72, n4, 2001, pp. 2062–2068.
23. Wilde (M.), Hategan (M.), Wozniak (J. M.), Clifford (B.), Katz (D. S.) et Foster (I.). – Swift : A language for distributed parallel scripting. *Parallel Computing*, vol. 37, n9, 2011, pp. 633–652.