Projet ROSES
Programme MDCO – Edition 2007

Livrable no D3.3
Definition of a wrapper adapted to blastfeed

Identification

Acronyme du projet	ROSES
Numéro d'identification de l'acte attributif	ANR-07-MDCO-011-01
Coordonnateur	Paris 6
Rédacteur (nom, téléphone, email)	YEH, 01 39 25 40 48, Laurent.Yeh@prism.uvsq.fr TRAVERS, 01 40 27 27 25, Nicolas.travers@cnam.fr
No. et titre	
Version	0.1
Date de livraison prévue	Date / t0+10
Date de livraison	31 Janv 2009

Résumé

L'objectif du wrappeur proposé dans le delivrable D3.3 est d'offrir une connexion aux sources de données stockées dans la plateforme de blastfeed. Un tel accès permet de stocker d'extraire et de stocker des résultats lors d'évaluation de requêtes données sur les flux RSS suivant l'architecture globale du projet RoSeS. Afin de pouvoir accéder à une multitude de sources, le wrappeur permet

Par rapport à la prévision initiale qui est d'accéder à la plateforme de blastfeed, nous avons du développer un autre type de wrappeur. En effet, la plateforme blastfeed est basée sur le serveur Xylème qui n'est plus maintenue suite à la faillite de la société Xyleme. Nous avons donc en plus d'un wrappeur accédant à une source Xylème du développer un autre wrappeur XQuery accédant à une source du type eXist et décrit dans ce delivrable D3.3.

Introduction

Le contexte du flux RSS

Les flux RSS basés sur les concepts XML du W3C constituent une nouvelle approche pour la diffusion et le partage des informations dans le web. Elle s'appuie sur plusieurs normes telles que RSS 0.91 (Rich Site Summary), RSS 0.90 et 1.0 (RDF Site Summary), RSS 2.0 (Really Simple Syndication), et ATOM []. Cette technologie comble un manque dans le monde du web. En effet, elle est habituellement utilisée pour signaler les mises à jour des sites dont le contenu change

fréquemment, typiquement les sites d'information ou les blogs [Édi06] [17]. L'utilisateur peut s'abonner aux flux et ainsi être informé des dernières mises à jour sans avoir à se rendre sur le site. Un flux RSS est un document XML mis à disposition par un serveur Web. Un flux contient un ensemble d'éléments suivant une des normes susdite. Il est structuré comme une séquence d'éléments appelés des "items". Chaque item structuré contient une date de publication, un résumé et un lien vers une page Web. Le client s'abonne au flux via un lecteur de flux qui est présent dans la plupart des navigateurs (IE, Firefox). Le lecteur de flux rafraîchit ensuite régulièrement le flux selon la période spécifiée par le client en réinterrogeant le serveur web mettant à disposition le flux.

Du point de vue utilisateur, la consommation de ces données pose de nouveaux problèmes. En effet, l'utilisateur ne peut ou ne souhaite lire en temps réel l'ensemble des données provenant de ces flux. Un utilisateur peut être rapidement submergé par les données provenant de nombreux flux. Il doit pouvoir se faire aider par des outils qui lui synthétisent une information en tenant compte de la dimension temporelle. Un exemple de requête sur les flux est : «Quels sont les articles sur les jeux olympiques de Pékin durant les trois derniers jours?». Pour cela, il est nécessaire de conserver les flux sur une période de temps [1].

> *Objectif du wrappeur*

Pour enrichir les possibilités des requêtes offertes aux utilisateurs, il est nécessaire de stocker les flux récents dans une base de données. En effet, outre les requêtes standards offertes par tout lecteur de flux, le stockage des flux permet d'effectuer des requêtes plus riches et d'utiliser des possibilités d'interrogation comme XQuery. Par exemple : « Si le nombre d'items concernant l'A380 s'accroit deux fois plus durant les 3 derniers jours par rapport à la semaine, alors je veux être averti ». Un cache est donc nécessaire pour ne pas se limiter aux simples requêtes de "sélection/projection".

Ce rapport vise à décrire un wrappeur au dessus de SGBD/XML comme eXist. L'ensemble wrappeur vise à pouvoir y accéder via une technologie de médiation comme celui illustrée ici (XLive). Pour cela, le rapport décrit la technologie de médiation, puis dans une seconde partie, le rapport décrit les adaptateurs permettant d'accéder aux sources.

Médiation des sources de données

> A première vue, comme je ne connais pas bien le wrappeur, un plan possible pourrait être:
> - Into
> - Description de Blast Feed comme source de données
> - La mediation des sources relativement à l'architecture
>
Je peux te fournir une bonne partie sur la médiation de sources de données avec les TGV.
Je t'ai mis en attaché un TeX comme introduction/transition pour médiation avec TGV. (les références sont intégrées en bibTeX dans ref.bib)
J'ai aussi ajouté un autre sur les TGV avec les différentes parties importantes.

Une source accéder par un médiateur

Le système de médiation XLive intègre *via* des adaptateurs des sources de données variées (SQL, XQuery, Fichiers XML, LDAP ...). Le logiciel XLive est mis en logiciel libre et a été utilisé dans plus plusieurs projets : projets européens IST *WebSI* \cite{WebSI}, Satine \cite{Satine}, le projet ACI *SemWeb* \cite{SemWeb} et le projet ANR *PADAWAN* \cite{Padawan}.

L'objectif d'un médiateur est d'intégrer un grand nombre de sources hétérogènes disséminées. Il doit pouvoir intégrer les données pour répondre aux requêtes des utilisateurs. Une requête soumise par

l'utilisateur est traitée par le médiateur. Elle produit, grâce à un ensemble de transformations définies dans le médiateur, un ensemble de sous-requêtes qui sont alors transmises aux adaptateurs. Ces derniers communiquent ensuite avec les sources. Les résultats sont intégrés par le médiateur qui créé le résultat final correspondant à la demande de l'utilisateur.

Les architectures des bases de données fédérées ont progressivement convergé vers une vue unifiée proposée par DARPA I3 \cite{patil92darpa} et Gio Wiederhold \cite{Wiederhold1992-mediators-architecture}.

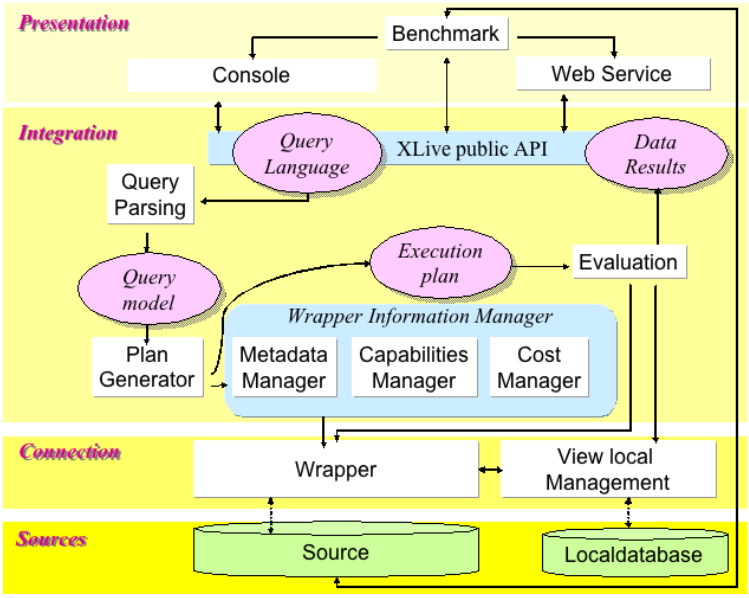

Figure X1 *: Principe de la déportation des commandes*

Suivant la figure ci-dessus, cette architecture se compose de trois niveaux:

- *Le niveau source* (connexion et sources sur la figure): comporte les différentes sources de données. Les sources de données communiquent avec le médiateur à l'aide *d'adaptateur* (*wrapper*) publiant de manière homogène les données de la source. L'adaptateur est chargé de (i)} traduire une requête exprimée dans le langage du médiateur en une requête exprimée dans le langage de la source, (ii) faire évaluer la requête par la source et (iii) renvoyer les résultats au médiateur.

- *Le niveau médiateur* (intégration sur la figure): comporte un ou plusieurs médiateurs permettant d'intégrer les données transmises par les adaptateurs} des sources. Il s'occupe de faire l'interface entre une requête utilisateur et les sources évaluant la requête suivant le modèle décomposition, recomposition, optimisation} cité précédemment. Il présente les données de manière homogène et centralisée à la couche supérieure, résolvant le problème d'hétérogénéité et de distribution des données.

- *Le niveau client* (présentation sur la figure): comporte l'application cliente pour accéder aux médiateurs (i.e. navigateur, interface graphique, API cliente).

L'ajout d'une nouvelle source à un système de médiation entraîne la définition d'un nouvel adaptateur pour la rendre accessible \cite{cluet98your}.

XLive propose des adaptateurs permettant la communication avec toutes sortes de sources de données : natives XML (Xyleme, eXist, Xhive), relationnelles (Oracle, MySQL, Microsoft Access),

fichiers XML, Web Services (Google API, Amazon).

Intégration du médiateur dans l'architecture globale de RoSeS

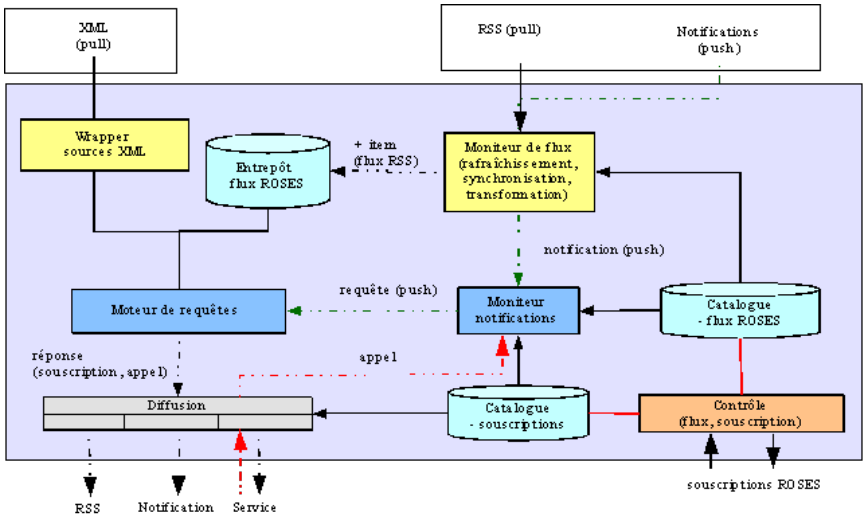

***Figure X2** : Adaptation de la source dans le wrappeur*

Suivant l'architecture globale (figure X2) du projet RoSeS, un wrappeur doit permettre d'accéder à la source d'un système de syndication.

Le médiateur XLive permet d'analyser les requêtes XQuery reçues pour les transformer et les envoyer aux sources. A partir de ce système, les fonctions de recherche de contenu et de classement de résultat sont accessibles dans une architecture de médiation, de manière uniforme et efficace.

Représentation des requêtes

Pour répondre à ces problèmes, nous proposons un modèle de représentation de requêtes que nous appelons *Tree Graph View* (TGV). Ce modèle est basé sur un ensemble de motifs d'arbre correspondant aux collections ciblées par les requêtes, ces motifs forment une représentation générique d'une interrogation sur une source. Par ailleurs, en intégrant les contraintes ainsi que les opérations complexes existantes dans le langage XQuery, nous pouvons représenter la quasi-totalité du langage (8 des 9 groupes de requêtes définis dans les « uses cases du W3C » sont vérifiés par les TGV).

Le TGV est alors traduit sous forme d'arbre algébrique composé d'opérateurs de la XAlgèbre [Xlive]. Ceux-ci manipulent des arbres DOM à l'aide de pointeurs sur valeurs précompilés. Les documents XML résultats sont analysés après leur sorties des adaptateurs (wrapper) dans un opérateur qui se nomme *QueryComposer*, dépendant de la source interrogée. Chaque document génère un XTuple qui est une instance de tuple contenant l'arbre DOM et les références nécessaire dans l'arbre d'évaluation de la XAlgèbre.

Un wrapper d'accès aux sources

Le rôle d'un wrappeur

Un autre sur les adaptateurs en général, avec DARPA I3, etc... et la description des adapteurs présents dans XLive. (je peux encore t'en rajouter, mais je pense que ca fera un peu trop)

Les systèmes de médiation ont été largement étudiées \cite

{wiederhold1992,manolescu2001,naacke1998,baru1999,bda2005}.
De tels systèmes de médiations \cite {wiederhold1992} fournissent une interface uniforme à une multitude de sources de données, en utilisant des médiateurs et des adaptateurs pour gérer respectivement la distribution et l'hétérogénéité.
Ces cinq dernières années, les systèmes de médiation \cite {baru1999,bda2005} ont utilisé XML comme format d'échange de données, et XQuery comme langage de requête commun.
Le système XLive \cite {bda2005} utilise le TGV comme modèle de représentation XQuery.

L'hétérogénéité des données est géré par des adaptateurs (wrappers) qui agissent comme des « traducteurs » entre le langage de requête natif (resp. le format natif de résultats) des sources de données et le langage de requête commun (resp. le format de réponse commun) utilisé dans l'architecture de médiation.
Le médiateur décompose la requête utilisateur en sous-requêtes envoyées adaptateurs, et recompose le résultat final.
De ce fait, le médiateur gère les aspects de distribution des données.

Wrappeur pour Xylème,

> *Alors, j'ai bien cherché, et je ne pense pas avoir quelquechose la-dessus précisément. Je peux donc pondre un texte, mais il ne fera pas 3 pages (tout au plus 1/2)*
> *Je décrirai la nécessiter de faire la traduction de requête en XyQL, la restructuration du TGV pour qu'il s'adapte, et la difficulté pour restructurer les résultats.*

Initialement, XLive utilisait Xylème pour l'interrogation de documents XML. Celui-ci proposant un stockage et une indexation de ses documents efficace, la solution a été retenue. Toutefois, le langage de requête *XyQL* demandait une couche supplémentaire d'analyse et de traduction des requêtes. Seul un sous-ensemble de XQuery était facilement traduisible. Il était possible de recomposer toutes les requêtes, mais le résultat dans Xyleme n'était alors plus optimal du aux nombreuses requêtes nécessaires pour l'évaluer.

Ci-dessous, vous pouvez voir la traduction d'une requête simple dans le langage XyQL :

for $b in collection("catalog")/catalog/book
where
  $b/price > 100
return
  <titre> {$b/title/text()} </titre>
	SELECT
	  Element ("titre", b/title/text())
	FROM b IN catalog/catalog/book
	WHERE b/price>100 ;

Wrappeur 2 pour eXist

eXist est un SGBD XML plus récent (version stable 2002) qui permet d'indexer des documents XML et d'interroger celui-ci à l'aide du langage XQuery. Grâce à l'interface simple d'eXist, la connexion avec XLive en est grandement améliorée.
En effet, un TGV représentant la requête XQuery peut être découpé et simplement traduit en XQuery sur la source de données. Les résultats sont alors récupéré par paquets (resultSet) grâce à l'interface d'application de eXist, ce qui facilite grandement les transferts de données et optimise les traitements au niveau du médiateur de données.

Le wrapper eXist a besoin de trois composants principaux pour fonctionner :

- **QueryComposer** : afin de transformer un résultat dans le modèle de données de la source (ici un document XML) vers le modèle de XLive (XTuple)
- **XLiveConnection** : afin de gérer la connexion entre le médiateur et la source de données. Il permet de créer des *statements* pour chaque requête.
- **XLiveStatement** : gère une instance de requête en interrogeant la source de données et en préparant les résultats vers le *QueryComposer*.

Ci-dessous, vous pourrez trouver l'API des trois composants du wrapper eXist :

Class QueryComposereXist

```
java.lang.Object
  └xlive.xalgebra.QueryComposer
      └xlive.wrappers.eXist.QueryComposereXist
```

Constructor Detail

QueryComposereXist

```
public QueryComposereXist()
```

Method Detail

parseQuery

```
public java.lang.String parseQuery()
```

Contrary to other Quercomposers, the parseQuery function does not transforms the XQuery query, since eXist understand the XQuery language.
Returns:
The query

---

initXLiveStatement

```
public void initXLiveStatement()
```

Creates the XLiveStatement from the XLiveConnectioneXist connection.
See Also:
`XLiveStatement`

Class XLiveConnectioneXist

```
java.lang.Object
  └xlive.mediator.connection.XLiveConnection
      └xlive.wrappers.eXist.XLiveConnectioneXist
```

Constructor Detail

XLiveConnectioneXist

```
public XLiveConnectioneXist()
```

Method Detail

connect

```
public void connect(java.lang.String theServer,
                    java.lang.String plogin,
                    java.lang.String ppassword,
                    java.lang.String addon)
             throws java.lang.Exception
```

Establish the connection to the eXist database.

Parameters:

`theServer` - server address

`plogin` - connection login

`ppassword` - connection password

`addon` - misc. connection information (like collection selection, index, etc…)

Throws:

`java.lang.Exception`

---

openXQuery

```
public org.xmldb.api.modules.XPathQueryService openXQuery()
                                     throws org.xmldb.api.base.XMLDBException
```

Creates an XQuery instance on the eXist database.

---

importMetadatas

```
public java.lang.String importMetadatas()
```

Import metadata from the eXist database, helps defining global metadatas on the XLive system.

Returns:

the metadatas

Class XLiveStatementeXist

```
java.lang.Object
  └xlive.mediator.connection.XLiveStatement
      └xlive.wrappers.eXist.XLiveStatementeXist
```

Constructor Detail

XLiveStatementeXist

```
public XLiveStatementeXist(XLiveConnectioneXist connection)
```

Initialization of the QueryManager and the session

Method Detail

init

```
public void init(java.lang.String tag,
                 org.xml.sax.helpers.DefaultHandler handler)
          throws org.xml.sax.SAXException,
                 javax.xml.parsers.ParserConfigurationException
```

Initialize parsing arguments and the SAXParser for the XML resultset

---

getSchema

```
public java.util.List getSchema()
```

Returns a list of the resulting schema.

---

executeQuery

```
public void executeQuery(java.lang.String query,
                         java.util.List<java.lang.Integer> localIDList)
                  throws org.xmldb.api.base.XMLDBException
```

Run the query execution on the eXist database.

Parameters:

`query` - the query to execute

`localIDList` - view local identifiers required

---

executeSimpleUpdate

```
public void executeSimpleUpdate(java.lang.String commands)
```

For update purposes, usable threw external applications.

---

hasNext

```
public boolean hasNext()
                throws org.xmldb.api.base.XMLDBException
```

Verify if there are other results in the ResultSet

---

parseNext

```
public void parseNext()
               throws org.xmldb.api.base.XMLDBException,
                      java.io.IOException,
                      org.xml.sax.SAXException
```

Use the SAX Parser on the current document to generate a SAX event that could be brought into the QueryComposereXist

---

close

```
public void close()
```

Close the session and finalize the ResultSet to the QueryComposer

API Publique d'un nœud et protocole d'accès aux nœuds

L'interface d'application de XLive propose les fonctionnalités suivantes :

Interface XLiveInterfaceMediator

```
public interface XLiveInterface
```

XLiveInterface is the interface for creating a mediator and executing XQuery queries.

The API interface for accessing the XLive mediator defines methods for initializing the mediator from a file describing connections and a file describing metadatas.

Author:

Clément Jamard clement.jamard@prism.uvsq.fr, Nicolas Travers nicolas.travers@prism.uvsq.fr

Method Detail

initialize

```
void initialize()
```

Initialize a new mediator without any connection nor metadatas

---

initialize

```
void initialize(java.lang.String connectionFile,
                java.lang.String metaDataFile)
                throws xlive.xliveapi.XLiveException
```

Initialize a mediator from a connection and a metadata file.

Parameters:

`connectionFile` - the connection file

`metaDataFile` - the medadata file

`exceptionManager` - exception manager for the api

---

getMetadata

```
java.util.List<org.w3c.dom.Node> getMetadata()
```

Return mediator metadatas as pathset trees.

Returns:

a list of pathset tree

---

importMetadata

```
void importMetadata(java.util.List<java.lang.String> sources)
                    throws xlive.xliveapi.XLiveException
```

Retrieve metadatas from selected sources.

Parameters:

`sources` - selected sources

---

registerConnection

```
void registerConnection(java.lang.String xliveConnection,
                        java.lang.String name,
                        java.lang.String login,
                        java.lang.String querycomposer,
                        java.lang.String sourceurl,
                        java.lang.String password,
                        java.lang.String baseURI)
                        throws xlive.xliveapi.XLiveException
```

Register a new connection.

Parameters:

`name` - the connection name

`xdbcurl` - the connection protocol

`sourceurl` - the source url

`login` - the connection login

`password` - the connection password

`database` - the used database

---

getConnection

```
java.util.List<xlive.mediator.connection.XLiveConnection> getConnection()
```

Return registered connections of the mediator as `XLiveConnection` objects.

Returns:

the list of registered connection

See Also:

`XLiveConnection`

---

getQuery

```
java.lang.String getQuery()
                          throws java.lang.NullPointerException
```

Return the current query processed in the mediator.

Returns:

the XQuery query.

---

setQuery

```
void setQuery(java.lang.String query)
              throws xlive.xliveapi.XLiveException,
                     xlive.views.exception.XLiveViewException
```

Prepare a query for execution by creating the execution Plan

Parameters:

`query` - the query

---

setSource

```
void setSource(java.util.List<xlive.mediator.connection.XLiveConnection> source)
```

Set a list of sources for the current query to process. The list of sources is a list of `XLiveConnection` objects.

Parameters:

`source` - a connection list.

---

getQueryModel

```
xlive.querymodel.QueryModel getQueryModel()
```

Return the Model of the curent query processed in the mediator.

Returns:

the query QueryModel.

---

getPlan

```
xlive.xalgebra.xoperator.XOperator getPlan()
```

Return the algebric plan (expressed in XAlgebra) as a tree composed of `XOperator`.

Returns:

the last operator of the plan.

See Also:

`XOperator`

---

execute

```
java.util.List<java.lang.String> execute()
                                          throws xlive.xliveapi.XLiveException
```

Launch the execution of an algebric Plan (include the evaluation of the `XPlan`).

Returns:

the XML results list

---

evaluate

```
void evaluate()
            throws xlive.xliveapi.XLiveException
```

Evaluate the execution plan corresponding to the XQuery query. Results are accessed as `XTuple` with next() or as XML data with nextAsString()

---

hasNext

```
boolean hasNext()
              throws xlive.xliveapi.XLiveException
```

Check if more results are available in the execution Plan.

Returns:

true if the plan contains more results, false otherwise

---

next

```
xlive.xalgebra.XTuple next()
                         throws xlive.xliveapi.XLiveException
```

Get the next result as a `XTuple`.

Returns:

the next result

nextAsString

```
java.lang.String nextAsString()
                                throws xlive.xliveapi.XLiveException
```

Get the next result as a String.

Returns:

the next result

Pour utiliser le médiateur, il faut pour cela initialiser les connexions à l'aide de **'initializeConnection'** qui utilisera le fichier de configuration paramétré pour se connecté à la source de données eXist.
Ensuite, une requête peut être utilisée à l'aide de la fonction **'setQuery'** et évaluée grâce à **'evaluate'**. Enfin, les résultats sont disponibles grâce à la fonction **'nextAsString'**.

Conclusion

Annexe