Ersch. in: ICWE'08: Eighth International Conference on Web Engineering : proceedings : 14-18 July 2008, Yorktown Heights, New York, USA / Daniel Schwabe ... (eds.). - Piscataway, N.J. : IEEE, 2008. - S. 162-174. - ISBN 978-0-7695-3261-5

http://dx.doi.org/10.1109/ICWE.2008.13

Database-Driven Web Mashups

Andrei Vancea Michael Grossniklaus Moira C. Norrie
Institute for Information Systems
ETH Zurich
CH-8092 Zurich, Switzerland
{*vancea,grossniklaus,norrie*}*@inf.ethz.ch*

Abstract

In most web mashup applications, the content is generated using either web feeds or an application programming interface (API) based on web services. Both approaches have limitations. Data models provided by web feeds are not powerful enough to permit complex data structures to be transmitted. APIs based on web services are usually different for each web application, and thus different implementations of the APIs are required for each web service that a web mashup application uses. We propose a database-driven approach to web mashups that supports integration at the database level and enables mashup developers to work with a uniform abstract model and have direct access to powerful features of database systems. We describe how we have implemented this approach based on an object-oriented database system with a rich object model and a generic proxy mechanism for data integration and synchronisation.

1. Introduction

A web mashup is a web application that combines data from multiple web applications in order to create a new application [18]. In a web mashup application, the content is usually generated by web feeds, screen scraping or by calling a public programming interface. Most of these programming interfaces use web service technology [1]. Currently, four types of web mashups—consumer mashups, data mashups, enterprise mashups and business mashups—can be observed. Consumer mashups combine data from different web sites and use a simple unified graphical interface to display the combined information. An example of such a mashup would be a web application that integrates digital pictures from flickr.com and displays them on the Google map interface using the images' geographical tags. Data mashups are used to combine more than one data source into a single source such as, for example, several news feeds being combined into a single feed. An enterprise mashup uses the general mashup techniques within a company's own internal applications. It usually combines data from both internal and external sources. Finally, a business mashup is a combination of all of the above that makes the result available for a business application.

In this paper, we argue that current approaches to the development of web mashup applications lack a powerful data model for data interchange. As mentioned above, in most cases, the content is generated using feeds or an application programming interface (API) based on web services. In the former case, the data models provided by web feeds (RSS or Atom) lack flexibility and do not permit complex data structures to be transmitted. On the other hand, while the API approach is flexible, every web application tends to have a different API. As a consequence, there must be a different implementation of the APIs for each web service that a web mashup application uses.

We propose a database-driven approach to web mashups that allows data integration and mashup logic to be managed within a database. This enables developers of web mashup applications to work with a uniform abstract model and to have direct access to powerful features of database systems such as declarative querying, constraints, triggers and dynamic updates as well as persistence. By using an object-oriented database system, data exchange is based on objects which can be arbitrarily complex.

We have implemented a database-driven web mashup architecture using an object-oriented database system that is based on the OM data model [6, 7]. The *collection* is a central concept in the OM model for managing semantic groups of objects, and we show how it provides a flexible basis for dealing with data integration and synchronisation issues. A generic proxy mechanism [19] was used to manage communication and synchronisation between the web mashup application and its data sources, thereby enabling participants to exchange objects and collections of objects directly from their local databases. The final web site can then be built on top of the database using a content management system.

We begin in Sect. 2 with a general discussion of web mashup requirements and existing approaches. Section 3 gives an overview of the generic proxy mechanism. In Sect. 4, we then present our database-driven web mashup architecture. This is followed in Sect. 5 with an outline of how an application can be developed using our approach. Section 6 discusses some issues related to the efficiency and the development process of web mashup applications using our approach. Finally, some concluding remarks are given in Sect. 7.

2. Background

The term mashup has its origin in the music culture, where a mashup is a remix created by combining different music compositions [4]. In a similar way, a web mashup combines data from different web applications in order to offer users an integrated experience. Usually, the content is provided by web feeds such as RSS and Atom, or by using a third party programming interface based on web service technologies. As mentioned previously, currently available web mashups can be classified into four categories—consumer mashups, data mashups, enterprise mashups, and business mashups [14] which are a combination of the other three. We now examine each of the three basic categories in turn and the support that is currently offered to create such mashups.

Consumer mashups combine data from different web sites and use a simple unified graphical interface to display the combined information. These are the most common form of web mashups found today. The ProgrammableWeb [18] site stores more the 1800 consumer web mashups. Most of them are non-commercial experiments, created in an ad-hoc manner. The site offers more than 200 different APIs for creating and combining types of applications such as search engines, weather data, instant messaging, blogs and RSS aggregators. Many commercial web sites provide interfaces that allow applications to be written using the data offered by the site [4]. Amazon.com was one of the first sites that released a free programming interface. Nowadays, eBay, Flickr, Google and YouTube also offer powerful programming interfaces. The term web mashup was first used to refer to the housingmaps.com site which displays rentals and sale adds from craiglist.com in Google Maps.

Data mashups are used to combine more than one data source into a single source. For example, more than one RSS feed can be combined into a single one. There are many tools on the market that allow data mashups to be created in a simple manner. For example, Yahoo Pipes [15] is a web application that permits users to build aggregate web feeds by integrating data from different sources using a very intuitive graphical user interface. Microsoft offers a very similar application, called Microsoft PopFly [17]. The Google Mashup Editor [13] is another on-line application used for creating such mashups.

An enterprise mashup combines external web data with the internal data of an organisation [2]. For example, an enterprise mashup for a particular organisation could combine public data that refers to the prices of the properties in different areas of a city with the internal data about properties owned by that organisation. JackBE Presto [16] is a mashup platform designed for creating enterprise mashups.

Recently, web mashups have also become a topic of interest to the research community. MashMaker [3] is a web-based tool that can build mashups by graphically composing and integrating data from different sources. Even though it is not exclusively intended to build web mashups, the approach is very similar to the approaches found in the commercial products mentioned above. A similar approach is Mash-o-matic [5], a utility that combines different information fragments in order to generate input data for mashups. Based on this input data, it also generates the code for the final mashup system that can then be deployed. Their system uses an application called Sidepad for collecting and organising data. The information fragments are extracted by the mashup developer from different external sources.

Another approach is taken by Yu et al. [20] who have developed a framework for integration at the presentation layer. Instead of trying to combine the application logic and data, the framework rather integrates their graphical front-ends. Since the framework follows a model-driven approach, an application built with the framework is specified in terms of a *compositional model* and the middleware. The compositional model represents the logic that coordinates the components at runtime, while the *middleware* handles communication between the components in an event-driven manner.

Some research projects focus on the integration of web services. One such approach is Mashup Feeds [11], a system that enables developers to create new feeds by integrating web services. Feeds are expressed as continuous queries over the existing web services and feeds. In order to create the mashup application, a developer must specify a collection-based stream operator that performs the extraction of history-based tracking information and summaries. This operator is composed from a sequence of sub-operators such as *subscribe*, *join*, *select* and *map*. The Web Mashup Scripting Language (WMSL) [10] is a new scripting language that enables end-users to quickly define mashups that integrate two or more web services. This task is accomplished by adding metadata and JavaScript code directly to the HTML source. The metadata is specified in the form of mapping relations that are then used to reconcile the structural, syntactic or representational mismatches between data models.

Other approaches are suited to the development of enterprise web mashup applications. Damia is an enterprise data integration service where business users can create and catalogue data feeds of high value for consumption by situational applications [2]. The system provides a user interface that allows data mashups to be created as data flow graphs by the user. Further, the system offers a server that contains the execution engine as well as a powerful API used for executing and managing mashups. Finally, in Thor et al. [12], a framework architecture is presented that can be used for the development of dynamic data integration mashups. The framework consists of several components such as query generation and on-line matching as well as additional data transformation modules. The proposed architecture supports interactive and sequential result refinement. The system uses wrappers for transforming external data into a self-describing XML structure.

What all these approaches have in common is the fact that the data models used for data interchange are usually relatively simple. Applications developed with tools like Yahoo pipes or Microsoft PopFly are usually used to aggregate web feeds. The data models offered by these web feeds are not powerful enough to support more complex data structures. While web services allow complex structures to be interchanged, usually each participant has its own API and so the application must have different implementations for each such web service based API. Object-oriented database systems support semantically richer data models. Usually such models support object type inheritance and also have strong support for managing collections of objects. We believe that webmashup application development could profit from these concepts.

The subscription-based approach used by the web feeds can be adapted and used with the data model supported by an object-oriented database system. Thus, similar to the way subscription is used with web feeds, a client database can subscribe to some of the objects and collections offered by a server database system. The architecture takes care of synchronisation between the client database and the server database. More than that, bi-directional communication can also be supported. Thus, both the client and server can update shared objects and collections and have these changes propagated.

In the remaining sections, we describe the details of our approach in terms of a specific database-driven architecture that we have developed and show how it can be used to develop web mashup applications. Our architecture was developed using OMS Avon, an object-oriented database system based on the OM data model [6, 7]. OMS Avon is implemented as a semantic data management layer on top of db4o [8] to support features such as role modelling, associations, constraints, triggers and a declarative query language in addition to the basic features required for object persistence offered by db4o.

For the sake of simplicity, we assume in the following sections that all participant databases in our web mashup architecture are implemented using OMS Avon. In practice, it is more likely that the data sources would be based on a variety of technologies such as relational or XML database systems. As is common in integration architectures, wrapper components would be required to map to and from the global model, which in our case is the OM data model. We omit these wrapper components in the description of our architecture to enable us to focus on the basic mechanisms for data integration and synchronisation.

3. Generic Proxy

The communication and synchronisation mechanisms in our web mashup architecture were built using a generic proxy mechanism [19] which we describe in this section. Our goal was to provide a general mechanism that can be embedded in database management systems in order to support the integration of external heterogeneous data sources. This means that it had to be able to support the different categories of data integration systems that have been proposed to meet widely varying requirements and different levels of heterogeneity. In our system, the generic proxy mechanism is used to enable the communication between the participants of a mashup application. This is done by having a bi-directional synchronisation process of the two different views of an OM object stored in different database systems. In the OMS Avon object-oriented database system, collections of objects are internally represented as objects, so the generic proxy mechanism can be used to synchronise collections as well as individual objects.

The two general categories of data integration systems common nowadays are based on two different paradigms. The *virtual view* approach provides a virtual, unified view of data held in a number of data sources in order that users and applications can query across heterogeneous data sources. Mechanisms are provided to allow global queries expressed over the unified view to be decomposed into multiple sub queries which are sent to the different external data sources for processing, and the results of these local queries are then combined according to the unified view. The module that decomposes the global query and combines the results of the local queries is usually referred to as a *mediator*.

In contrast, a *materialised view* approach provides an actual repository of integrated information available for querying and analysis. Data is extracted from the data sources and stored in a central repository which can be queried directly without any need to refer to the external data sources. *Data warehouse systems* are an important category of systems based on the materialised view approach. Since the data from the information sources is transferred to

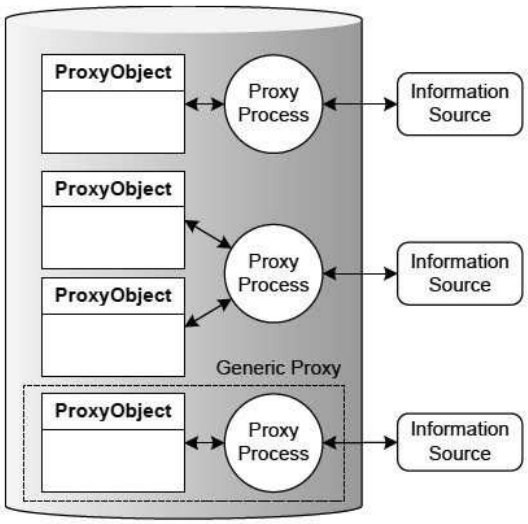

Figure 1. Generic proxy

the central repository before the execution of queries, this solution is sometimes referred to as an *eager* or *in-advance* approach to data integration.

In our approach, the integration of external information sources is done using a *generic proxy*. A generic proxy consists of two parts, the *proxy object* and the *proxy process*. The proxy object represents the database view of the external data source. The data from the external source is cached locally, similar to the materialised view approach. Queries can be executed locally without any communication to the external source. The running proxy process handles the communication between the database and the outside world. The architecture of the generic proxy approach is illustrated in Fig. 1. The synchronisation between the database view of the information source and the external information source is done automatically by the database management system in a transparent way.

We have defined a proxy programming interface that allows the user to specify how a proxy object interacts with an external source. The user has to write different implementations for different types of external sources. The proxy processes are created from particular implementations of the proxy interface.

A proxy process must handle the bi-directional communication between the database management system and the external source. When the proxy object is changed, the database system, using the proxy process, sends the modifications to the information source. At the same time, when the external information source is changed, the database system is notified by the proxy process. Having a running proxy process for each proxy object is clearly not a feasible solution. We therefore chose to map more than one proxy object to a single proxy process. By using the proxy programming

```
procedure createGenericProxy(name, args)
begin
  pobj = createProxyObject;

  found = false;

  //finding a suitable proxy process
  for process: getProxyProcesses(name)
  begin
    if process.accept(args) then
    begin
      associate (process, pobj);
      found = true;
      break;
    end;
    if (!found) then
    begin
      process =
        createProxyProcess(name, args);
      associate(process, pobj);
    end;
  end;
end;
```

Figure 2. Proxy object creation

interface, the user can specify how the mapping of proxy objects to proxy processes is done for particular types of proxies.

Generic Proxy Creation When a user wants to create a new proxy object, they must specify the name of the proxy and also the list of arguments that are needed in order to initialise the generic proxy.

Figure 2 illustrates how a new generic proxy is created. First, a new proxy object is created and stored in the database. Afterwards, the proxy object must be associated with an existing or newly created proxy process. This association is performed using a chain of responsibility approach. All of the existing proxy processes pertaining to the current proxy type are asked to accept the newly created proxy object using the *accept* call. Different proxy types can have different implementations of the accept() method. The proxy object is associated with the first process that accepts it. If no such process is found, a new one is created and associated with the proxy object. The association between the proxy object and the proxy process cannot be changed at a later time.

The Proxy Processes The proxy processes handle the bi-directional communication between the database and the external information sources. As already discussed, each proxy process has one or more proxy objects associated with it and the association between a proxy object and a

proxy process is established when the generic proxy is created.

A proxy process monitors the external information source and notifies the database when the external source is changed. As a result of the notification, the database system will schedule the corresponding proxy object for synchronisation.

An important issue related to the design of the proxy processes is the way in which a proxy object is referenced when the proxy process wants to notify the database system that a particular external information source was changed. A mechanism was added that allows a proxy object to be referenced with the generic identifier of the corresponding external information source. A generic identifier is a value of an arbitrary type that can be used to refer to an external information source in the outside world.

We offer a programming interface that allows developers to write different implementations of proxy processes. These implementations usually perform some kind of schema mapping and also handle the communication between the database and the external information source. We currently have different implementations of the generic proxy that enable the integration of data from different external sources. One such implemenation enables the integration of data from XML files. In this application, the proxy objects have associated elements stored in an external XML file. The schema mapping between the OM model and the XML model as well as the access to the external file is done in the proxy process.

Synchronisation One of the most important aspects of the generic proxy mechanism is the synchronisation between proxy objects and external data sources. We maintain a FIFO list that contains the proxy objects that are scheduled for synchronisation with their external information sources. A proxy object is added to this list in one of the following cases.

a) *If the value of one of its attribute is modified.* A change in the proxy object must be followed by a change in the external information source, so the proxy object must be synchronised.

b) *As a result of the modification of the external information source.* We have already described above the way in which the database system is notified when the external source is changed.

c) *At fixed time intervals.* Proxy objects may be set to be periodically resynchronised.

The generic proxy mechanism shares many similarities with integration systems that use the materialised view approach. In both approaches, the data from the information

```
procedure synchronise(proxyObject)
begin
  proxyProcess =
    proxyObject.getAssociatedProcess();
  remoteObject =
    proxyProcess.readRemoteObject();
  result = merge(proxyObject,
                 remoteObject);
  if (proxyProcess.setRemoteObject(result)
      == OK)
  then
    proxyObject = result;
  else
    proxyObject = remoteObject;
end;
```

Figure 3. Proxy object synchronisation

sources are extracted and stored in the database, allowing the queries to be evaluated without any need to refer to the external information sources. The major difference between the generic proxy mechanism and other systems is the architectural design. Most of the other systems are constructed using a middleware approach. A layer is added between the database and the information sources to handle the monitoring and integration of the external sources into the database. The generic proxy mechanism, however, is situated inside the database management system which allows the synchronisation between the database and the external sources to be performed automatically by the system in a transparent way. The proxy processes handle both the monitoring and the integration of the external information sources.

Another important difference between the generic proxy mechanism and the classic approaches—materialised and virtual views—is the fact that our mechanism may permit the client of the integration server to modify the external information sources. As Fig. 3 illustrates, the current state of the information source is compared with the value of the proxy object representing the database view of the information source during the synchronisation process. A new proxy object is then constructed and the value of the information source is changed.

By using the generic proxy mechanism, the synchronisation between the information sources and the database system is done automatically, in an efficient manner, when the information source is changed or when the value of its proxy object is modified. The system does not guarantee that the client will work with the latest versions of the information sources, but the synchronisation is usually done within a reasonable amount of time. We think that our solution is a good compromise between the real-time synchronisation offered by the mediated approach and the periodic synchronisation mechanism of the data warehousing approach.

4. Web Mashup Architecture

Our web mashup architecture allows a web application to share object-oriented database entities—objects and collections of objects—directly from its local database and other participants to subscribe to the entities provided by the application through the use of the generic proxy mechanism. Figure 4 illustrates this architecture. The two main components of our architecture are the *mashup application server* and the *information provider*. An information provider shares a list of objects and collections. A mashup application server can subscribe to one or more entities shared by the information provider. The communication between the participants is done using the SOAP protocol. Our architecture assures the bi-directional synchronisation between the objects and collections stored in the information providers and their materialised views of the mashup application servers. The web sites are built on top using content management systems. We say that a mashup application server is a client of an information provider if it is subscribed to at least one entity shared by the information provider. It should be noted that the mashup application server and the information providers are actually roles, and a participant can simultaneously be both a mashup application server in relation to some participants and an information provider for others.

An information provider contains middleware on top of the database system, that handles the communication with its clients and also the synchronisation process. The main components of this middleware are the *Subscription Manager*, the *Object Registry*, the *Collection Registry*, the *Object Dispatcher*, the *Collection Dispatcher*, the *Session Manager* and the *Web services interface*. In order for an object or a collection to be shared with the outside world, they must first be registered with the object registry and collection registry, respectively. The mashup application server uses the generic proxy mechanism to access objects shared by an information provider. The subscription handler takes care of the clients' subscriptions to the information provider. Communication between the information provider and its clients is done using a web service interface.

Let us consider an example where we want to develop an on-line web store application using our web mashup system. Apart from the usual features expected from an on-line store application, new products may be added to the on-line store directly by the producers using a well defined business-to-business interface. The server should also allow clients to be notified when new products are added or removed. For each producer, the on-line store will register a collection containing objects of type product. Using our mashup architecture, producers can subscribe to that collection. When a producer wants to sell a new product using the on-line store, they add an element to that collection. Using the synchronisation mechanism, the on-line store will receive the products and perform the necessary operations to display the products on the web site. The on-line store will also register collections that contain the list of all products and the list of offers.

Object Registry The object registry is used for registering objects in the information provider. An object must be registered in order to be available to the clients. After registration, a client can subscribe to the object. The interface of the ObjectRegistry class is given in Fig. 5. The method registerObject() is used for registering objects. The objectName argument represents the name that will be used to refer to the newly registered object from the clients. objectId is the internal database identifier of the object. The OM model supports multiple instantiation which means that objects may have multiple types and types represents the list of types of the specified object that can be accessed by clients. The getRegisteredObjects() method returns the collection of the registered objects. isRegistered() returns true if there is a registered object with the specified name. Finally, method getTypes() returns the types of the specified object.

```
public class ObjectRegistry {

  public void registerObject(
    final String objectName,
    String objectId,
    Collection<String> types) { ... };

  public Collection<String>
    getRegisteredObjects() { ... };

  public boolean isRegistered(
    final String objectName) { ... };

  public Collection<String> getTypes(
    final String objectName) { ... };

  public long getObjectVersion(
    final String objectName) { ... };

}
```

Figure 5. ObjectRegistry interface

Each registered object is associated with a version. The version is a positive integer which is incremented every time the object changes. Versions are used during the synchronisation process. Each time the object is modified, either from within the database or as the result of a change performed by one the clients that subscribed to the object, the version

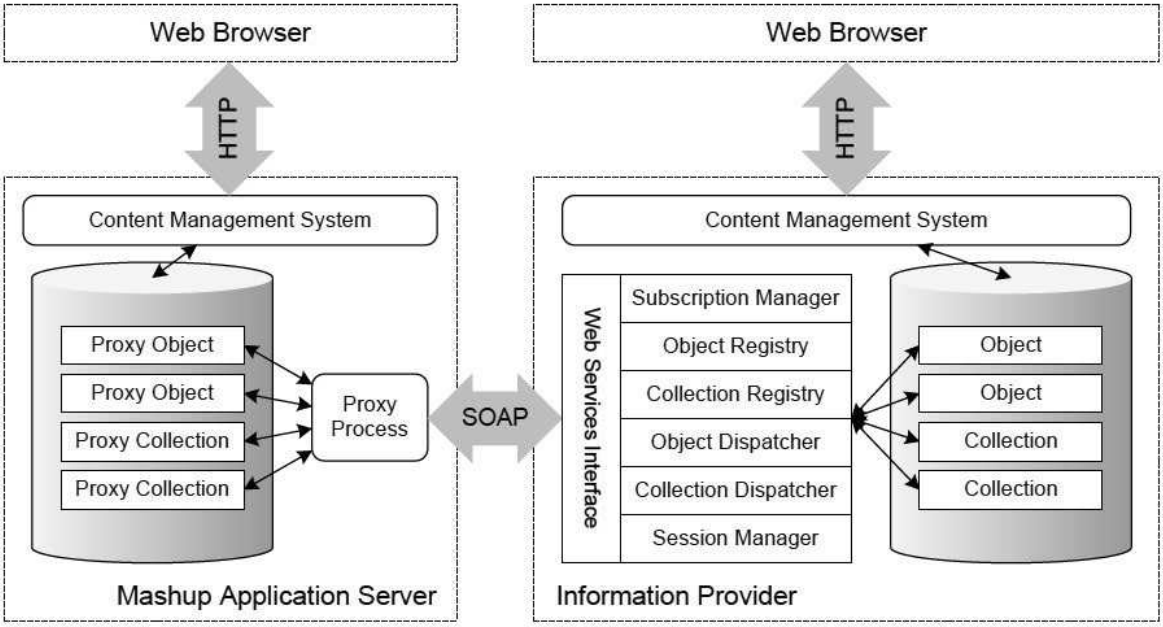

Figure 4. Web mashup architecture

is incremented automatically. Method getObjectVersion() returns the version of the specified object.

As mentioned above, each registered object can have multiple types. Referring to a type by its name only is not enough, because the information provider and client could have different types with the same name. For example, the product type on the server could have the attributes name as string and price as real while the product type on the client could have different attributes, for example prod_name as string and cost of type integer. In order to solve this issue, types are not identified by their name in our system, but rather using the type format. This derived type identifier is computed by applying a hash function to the serialised description of an object type.

Collection Registry The interface of the CollectionRegistry class is presented in Fig. 6. The method registerCollection() is used for registering a collection in the information provider. Method getRegisteredCollections() returns all registered collections, while the isRegistered() method checks whether a particular collection is already registered. Finally, getCollectionVersion() returns the version of the collection, which again will be used during synchronisation.

Session Manager In our approach, the communication process between an information provider and its clients is stateful. Communication sessions are handled by the session manager. Before interacting with the information provider, a client must first subscribe using the session manager. Figure 7 presents the interface of the session manager. Method createSession() is used to create a new session on the information provider and returns a session identifier. The session identifier is a string value that will be used during all communication with the client. closeSession() must be used when a session is no longer required. For each session, we keep, on the information provider side, a list of objects and collections that a client has subscribed to as well as the versions of these entities at the last synchronisation.

```
public class CollectionRegistry {

  public void registerCollection(
    final String colName,
    String colId,
    String type) { ... };

  public Collection<String>
    getRegisteredCollections() { ... };

  public boolean isRegistered(
    final String collectionName) { ... };

  public long getCollectionVersion(
    final String objectName) { ... };

}
```

Figure 6. CollectionRegistry interface

```
public class SessionManager {

  public String createSession() { ... };

  public void closeSession(
    String session) { ... };

  public boolean sessionExists(
        String session) { ... };

}
```

Figure 7. SessionManager interface

Subscription Manager As mentioned before, our web mashup interface is based on subscriptions. This means that clients can subscribe to the objects and collections provided by the information provider. Subscriptions are handled by the subscription manager. Its interface is presented in Fig. 8. The communication session discussed above is specified as the first argument of each method. Methods subscribe() and subscribeCollection() are used by a client to subscribe to an object or a collection, respectively. unsubscribe() is used when a client wants to unsubscribe from an information provider. The getSubscription() method returns the list of all subscriptions.

```
public class SubcriptionManager {

  public void subscribe(String session,
    String objectName) { ... };

  public void subscribeCollection(
    String session,
    String collectionName) { ... };

  public void unsubscribe(String session,
    String Name) { ... };

  public Collection<String>
    getSubscriptions(
    String session) { ... };

}
```

Figure 8. SubcriptionManager interface

Object Dispatcher The object dispatcher handles the synchronisation of the objects shared through the web mashup interface. It is connected through the web service interface to the generic proxy implementation of each client. The dispatcher has methods returning the latest values of the shared objects. It also permits the client to change the data of shared objects. Figure 9 gives an overview of the object dispatcher class. The method getAttributeValues() returns all values of the attributes pertaining to a particular type of the object specified as an argument. Note that the specified object has to be registered first. getAttributeValue() returns the values of a single attribute, while setAttributeValue() and setAttributeValue() are used for updating object values.

```
class ObjectDispatcher {

  public Map<String, Object>
    getAttributeValues(String objectName,
    String typeName) { ... };

  public Object getAttributeValue(
    String objectName, String typeName,
    String attributeName)  { ... };

  public void setAttributeValue(
     String objectName, String typeName,
     String attributeName,
     Object value) { ... };

  public void setAttributeValues(
    String objectName, String typeName,
    Object[] values) { ... };

  public Set<String>
    getChanges(String session) { ... };

}
```

Figure 9. ObjectDispatcher interface

The method getChanges() returns a list of objects or collections that have been changed since the last synchronisation. This method is used during the synchronisation process. Its implementation compares the current versions of the objects with the version stored in the session information for the specified session and returns the name of objects for which the two versions are different.

Collection Dispatcher The collection dispatcher handles the synchronisation of collections shared through the web mashup interface. Figure 10 contains the interface of the collection dispatcher. The method getChanges() returns the list of the collections that have been modified since the last synchronisation. getCollectionExtent() returns the list of elements contained in the collection specified as an argument. Each element contained in the collection will also be registered in the information provider. getElementsAdded() returns the list of elements added to the specified collection since the last synchronisation, while getElementsRemoved() returns the list of elements removed from the collection since the last synchronisation.

```
class CollectionDispatcher {

  public Set<String>
    getChanges(String session) { ... };

   public List<String> getCollectionExtent
       (String colName) { ... };

   public List<String> getElementsAdded
       (String session, String colName)
       { ... };


   public List<String> getElementsRemoved
       (String session, String colName)
       { ... };

}
```

Figure 10. CollectionDispatcher interface

Web Services Interface The web mashup interface handles the communication between the mashup server and the information providers. It was designed using the facade design pattern. It allows the mashup server to access the middleware of an information provider through a single unified interface. The web service interface uses Tomcat and the Apache web server in order to communicate with the clients. The web mashup interface communicates with the rest of the mashup server modules using remote method invocation (RMI).

Web Mashup Generic Proxy The communication part of the mashup application server is implemented using the generic proxy mechanism described in Sect. 3. As described before, a generic proxy is composed from the proxy object and the proxy process. The proxy object represents the materialised view of the entity to each subscribed client and the proxy process handles the synchronisation between the proxy entity and the associated entity from the information provider.

When a client subscribes to an object or to a collection shared by an information provider, a new generic proxy is created locally. The proxy object is then the local handle of the object from the information provider and contains all of its types and values. A proxy collection contains all of the elements from the collection of the information provider. When a new object is added to the collection, a new proxy object will be added to the proxy collection. The newly created proxy collection will refer to the new object from the collection. Additionally, a proxy process is created for each information provider to which a clients connects. Thus, two or more proxy objects which refer to objects from the same information provider will have the same proxy process. The implementation of the proxy process communicates with the information provider using its web service interface. The proxy process periodically synchronises the local view of the data stored in the proxy objects with the data on the information provider.

As discussed in Sect. 3, the synchronisation of the proxy objects is done using specific implementations of the proxy process interface. The proxy process implementation handles the schema mapping, the synchronisation and the communication with the external information source. Due to the fact that both the information provider and mashup application server use the same data model, schema mapping is not necessary and thus the proxy process only handles the communication with information providers and the synchronisation of the shared entities. The pseudo-code of the proxy process implementation for our web mashup architecture is illustrated in Fig. 11. In order to communicate with the information provider, a session must first be created. After session creation, it periodically checks if there were any changes performed on the information provider to the objects or collections to which the clients are subscribed, or if there are locally modified proxy objects. A change to an object means a modification performed to its types or values, while a change to a collection implies that new elements were added or some elements were removed. getChanges() returns a list of modified objects or collections, while getDirtyEntites() returns the names of the entity for each local proxy object of collections modified. Modified objects are then scheduled for synchronisation. The information provider is polled every t seconds, where t is a numeric constant that can be set by the user. Its default value is 60. Finally, the session is closed using the closeSession() call.

Synchronisation compares the values of the object on the information provider with the values of the associated proxy object. If both values have changed, the value from the information provider will be kept, otherwise the values of the attributes that were changed are preserved.

5. Application developement

In this section, we describe in detail how the on-line store example presented earlier could be implemented using our web mashup infrastructure. Let us assume that there are two on-line stores that use our architecture, two producers—a record label and publishing house—that sell products using the on-line store. Many on-line stores have affiliate programs which allow external sites to sell products offered by the on-line store, and receive a small percentage of the price of the products sold. The affiliate site must have access to the products offered by the on-line store. In this example, we introduce an aggregator that receives, using

```
procedure proxyProcess.run
begin
  session = infprov.createSession();
  while (not finished) do
  begin
    infprovChanges =
      infprov.getChanges(session);
    localChanges = getDirtyEntities();
    change = infprovChanges U localChanges;
    for i = 0 to changes.length do
    begin
      name = changes[i];
      synchronize(name);
    end;
    sleep(T);
  end;
  infprov.closeSession(session);
end;
```

Figure 11. Proxy process synchronisation

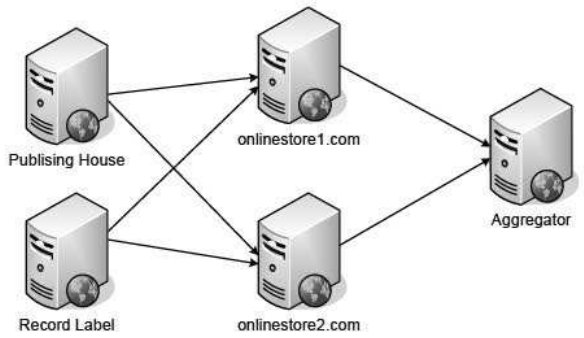

Figure 12. Example

our mashup architecture, products from both on-line stores and sells them using its own web site. The two producers are information providers in relation to the on-line stores, while the on-line stores are information providers for the aggregator.

The statements to configure the databases of both on-line stores are illustrated in Fig. 13. They are presented in OML, the native data definition, manipulation and query language of the OMS Avon database system. To support generic proxies, OML has been extended with additional statements that allow for their definition and management. First, we define the type product that will be used for storing products in the database. The quantity refers to the number of instances of a product that the producer has in stock. In order to use the facilities provided by the on-line store, both the producers and the clients must define this particular type in their database. For each producer, we create a collection in the on-line store database. Two collections are then registered. After the registration, the two collections—PublishingHouseProducts and RecordLabelProducts—are available for subscription to the two producers. In order to allow an affiliate site to access the list of products, we create the collections Products and Offers that are also registered on web mashup server middleware. The web site is then constructed on top of the so-configured database system.

```
create type product(name: string,
  producer: string, quantity: integer,
  price: real);

create collection PublishingHouseProducts:
  set of product;
create collection RecordLabelProducts:
  set of product;

create collection Products:
  set of product;
create collection Offers:
  set of product;

webmashup register collection
  PublishingHouseProducts
     as "PublishingHouseProducts";
webmashup register collection
  RecordLabelProducts
     as "RecordLabelProducts";

webmashup register collection
  Products as "Products";
webmashup register collection
  Offers as "Offers";
```

Figure 13. On-line store

The first producer is a publishing house that sells books using both on-line stores. The statements executed on the database of the first producer are presented in Fig. 14. The object type product must also be created in its local database. In order to add their products to an on-line store, the producer must subscribe to the collection provided by the on-line store. The `create proxy collection` statement creates a local proxy collection that refers to a collection provided by a remote web site. The first argument is the name of the proxy collection (OnlineStore1Books), while the second argument specifies the name of the proxy plugin, in our case the web mashup plugin. The next two arguments are the name of the site and the name of the shared collection. Afterwards, the objects that represent the products are created book1 and book2. Finally, the objects are added to the two proxy collections using the `insert` statement. As a result of the insertion, using the generic proxy mecha-

nism, the object will also be inserted in the two collections, both named PublishingHouseProducts, from the two servers onlinestore1.com and onlinestore2.com.

```
create type  product (name: string,
  producer: string, quantity: integer,
  price: real);

create proxy collection OnlineStore1Books
  webmashup "onlinestore1.com"
    "PublishingHouseProducts":
      set of product;

create proxy collection OnlineStore2Books
  webmashup "onlinestore2.com"
    "PublishingHouseProducts":
      set of product;

$book1 := create object;
dress $book1 with product (
  name = "George Orwell: 1984",
  producer = "FOO Publishing House",
  quantity = 100,
  price = 23.4
);

$book2 := create object;
dress $book2 with product (
  name = "Stephen Hawking:
                  A Brief History of Time",
  producer = "FOO Publishing House",
  quantity = 150,
  price = 30
);

insert in onlineStore1Books:
  [$book1, $book2];
insert in onlineStore2Books:
  [$book1, $book2];
```

Figure 14. Publishing house script

The script executed in the database of the second producer—the record label—is illustrated in Fig. 15. The script is similar to the one executed in the database of the first producer.

The aggregator integrates the products from both on-line stores and provides users with a web interface that allows products to be accessed from both on-line stores in a unified interface. The OML script used by the aggregator is illustrated in Fig. 16. First, the object type product is created. Afterwards, four proxy collections are created that will be used to synchronise the collections offered by the two on-line stores using our web mashup architecture. The web site giving access to the unified view of the web stores is then developed on top of the database and uses the defined

```
create type  product (name: string,
  producer: string, quantity: integer,
  price: real);

create proxy collection OnlineStore1CDs
  webmashup "onlinestore1.com"
    "RecordLabelProducts": set of product;

create proxy collection OnlineStore2CDs
  webmashup "onlinestore2.com"
    "RecordLabelProducts": set of product;

$cd1 := create object;
dress $cd1 with product (
  name = "Snow Patrol - Chasing Cars",
  producer = "FOO Record Label",
  quantity = 100,
  price = 50
);

$cd2 := create object;
dress $cd2 with product (
  name = "Coldplay - X&Y",
  producer = "FOO Record Label",
  quantity = 150,
  price = 30
);

insert in onlineStore1CDs:
  [$cd1, $cd2];
insert in onlineStore2CDs:
  [$cd1, $cd2];
```

Figure 15. Record label script

collections.

6. Discussion

In the previous sections, we have described the architecture of our database-driven web mashup system and how applications can be developed using our system. In this section, we will discuss some issues related to the efficiency and the development process of web mashup applications using our approach and compare our approach to some of the projects introduced in Sect. 2.

We are using the object-oriented data model OM for the exchange of data between the server and its clients. The OM data model supports multiple inheritance, multiple instantiation and multiple classification through built-in support for collections. In contrast to the relational model of data, the OM data model features directed associations as a first order concept to relate objects to one another. Finally, the OM data model also offers a rich set of constraints that raise its level of expressiveness and empower the developer

```
create type product(name : string,
  producer : string, quantity : integer,
  price : real);

create proxy collection onlineStore1Products
  webmashup "onlinestore1.com"
    "Products": set of product;
create proxy collection onlineStore1Offers
  webmashup "onlinestore1.com"
    "Offers": set of product;

create proxy collection onlineStore2Products
  webmashup "onlinestore2.com"
    "Products": set of product;
create proxy collection onlineStore2Offers
  webmashup "onlinestore2.com"
    "Offers": set of product;
```

Figure 16. Aggregator script

of an application to specify their requirements in more detail. We believe that building on a data model that is more versatile than the models currently used in web mashup application development will lead to better interoperability of such applications.

Currently, developing an application as described in Sect. 5 implies that both servers and clients have native database support for the OM data model. In our present implementation, both the clients and the server use OMS Avon, a semantic layer built on top of db4o, that supports the OM data model. As mentioned in Sect. 2, in practice, external information providers would most likely use a database system that uses another data model such as a relational system and a wrapper component would be required to map between the local data model and the OM model. This is common to most data integration architectures and a small number of such components can be provided to cater for the most common categories of information providers. We note that since the OM model is actually an extension of the E/R model of data such mappings are well-known and it is thus relatively straightforward to implement these mapping layers. While such a mapping layer introduces another level of indirection, we are convinced that the benefits of having a common, semantically rich data model outweigh this additional overhead. Finally, using a database system as a development platform also has other advantages. A database system offers technologies such as triggers and declarative query mechanisms that can also be used in the development of web mashups.

The web site will be generated from the objects stored in the database using a content management system. The web mashup system is separated from the site generation. This means that data integrated in the database using our mashup architecture could be used by a normal local database application in a transparent manner. Local database applications do not need to know that the objects stored in the database were gathered from external web sources.

We think that our system is especially suited to the development of enterprise web mashup applications. An enterprise web mashup application combines data from both external data sources such as the World Wide Web and internal sources. The generic proxy mechanism allows users to add proxies to different types of external data sources by creating different implementations of the generic proxy interface. For example, an application could be developed that combines data from a web server using the web mashup interface and local data from an XML file using an implementation of the generic proxy interface that integrates data from the XML files.

The communication between a mashup application server and an information provider is done periodically. As we have mentioned, communication is stateful. Reading data is associated with a session and thus the information provider knows the entities to which a mashup application server has subscribed. Using this information, communication between the information provider and its clients becomes quite efficient. Communication is initiated by the mashup application server requesting the list of entities that have been modified since the last synchronisation. From the list of objects and collections to which the client is subscribed, the information provider extracts the ones that have been modified. This on-demand synchronisation leads to an efficient system since entities are not exchanged frequently between the mashup server and the information providers.

Many of the current research projects related to web mashup technologies handle integration in the presentation layer, for example Yu et al. [20]. In our approach, the integration is done at the data level. We believe that the two kinds of systems are suited to different types of applications. Systems like the one described in Yu et al. [20] are suited to applications in which the user expects a real-time notification of modifications performed in one of the components. The above-mentioned system has an event-driven middleware that handles the real-time communication between the components. The volume of data exchanged in this kind of system is assumed to be low. Because the synchronisation between the client and server is done periodically, our system does not offer a real-time notification mechanism and is, therefore, better suited to applications that do not require real-time notifications. While changes are also propagated within a reasonable amount of time, the volume of data that is exchanged can be very large. As mentioned above, the server maintains a list of entities to which each client is subscribed and thus communication can be done efficiently even for large volumes of data. For example, in the on-line store application presented in this paper, change

propagation is not required to be done in real-time. After a new product is added to an on-line store, it is reasonable to expect that the aggregator will receive the information about the new product within a couple of minutes.

Using the approach described in [11], a user can create new integrated feeds as continuous queries over existing web services and feeds. Using our system, it is possible to create a new object or collection from two or more existing entities that are shared by other participants. This could be accomplished using the trigger mechanism to be notified when a proxy object or collection is changed and using the data manipulation language to update the resulting object or collection. The newly created entities can then be published to the web mashup server. As trigger and query mechanisms are natively supported by database systems, one can create powerful applications that combine two or more external sources into a single entity.

WMSL [10] uses mapping relations in order to reconcile the syntactic mismatches between data models. A similar problem could also arise in applications that are built using our approach. A client could import object from different servers that represent the same concept, but are represented using different schema. However, because we use the database system as a development platform, the schema mismatching could be solved directly in the database, using common schema matching algorithms [9].

7. Conclusions

We have shown how web mashups can be supported by integration and synchronisation at the database level. In particular, we have presented a web mashup architecture based on the object-oriented database system OMS Avon which offers full database functionality such as constraints, triggers and a declarative query language on top of the open source object database db4o. Importantly, it also offers a semanticaly rich object-oriented data model OM which supports concepts such as collections and associations as well as support for role modelling through multiple classification and multiple instantiation. We also presented a generic proxy mechanism for data integration and synchronisation, showing how it could be used to support a database-driven web mashup architecture.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Sevices: Concepts, Architectures and Applications*. Springer Verlag, 2004.

[2] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh. Damia: A Data Mashup Fabric for Intranet Applications. In *Proceedings of International Conference on Very Large Data Bases (VLDB'07), Vienna, Austria*, 2007.

[3] R. J. Ennals and M. N. Garofalakis. MashMaker: Mashups for the Masses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07), Beijing, China*, pages 1116–1118, 2007.

[4] I. R. Floyd, M. C. Jones, D. Rathi, and M. B. Twidale. Web mash-ups and patchwork prototyping: User-driven technological innovation with web 2.0 and open source software. In *Proceedings of Annual Hawaii International Conference on System Sciences (HICSS'07)*, pages 86–95, 2007.

[5] S. Murthy, D. Maier, and L. Delcambre. Mash-o-matic. In *Proceedings of the ACM Symposium on Document Engineering (DocEng'06), Amsterdam, The Netherlands*, pages 205–214, 2006.

[6] M. C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *Proceedings of International Conference on the Entity-Relationship Approach, Arlington, TX, USA*, pages 390–401, 1994.

[7] M. C. Norrie. Distinguishing Typing and Classification in Object Data Models. In H. Kangassalo, H. Jaakkola, S. Ohsuga, and B. Wangler, editors, *Information Modelling and Knowledge Bases VI*, pages 399–412. IOS Press, 1995.

[8] J. Paterson, S. Edlich, H. Horning, and R. Horning. *The Definitive Guide to db4o*. Apress, 2006.

[9] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal: Very Large Data Bases*, 10(4):334–350, 2001.

[10] M. Sabbouh, J. Higginson, S. Semy, and D. Gagne. Web Mashup Scripting Language. In *Proceedings of the International Conference on World Wide Web (WWW'07), Banff, Alberta, Canada*, pages 1305–1306, 2007.

[11] J. Tatemura, A. Sawires, O. Po, S. Chen, K. S. Candan, D. Agrawal, and M. Goveas. Mashup Feeds: Continuous Queries Over Web Services. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07), Beijing, China*, pages 1128–1130, 2007.

[12] A. Thor, D. Aumueller, and E. Rahm. Data Integration Support for Mashups. In *Proceeding of the International Workshop on Information Integration on the Web, Vancouver, Canada*, pages 104–109, 2007.

[13] `http://editor.googlemashups.com`. Google Mashup Editor.

[14] `http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)`. Wikipedia: Mashup (Web Application Hybrid).

[15] `http://pipes.yahoo.com`. Yahoo Pipes.

[16] `http://www.jackbe.com`. JackBE.

[17] `http://www.popfly.com`. Microsoft PopFly.

[18] `http://www.programmableweb.com`. ProgrammableWeb.

[19] A. Vancea, M. Grossniklaus, and M. C. Norrie. Generic Proxies – Supporting Data Integration Inside the Database (Poster). In *Proceedings of International Symposium on Distributed Objects and Applications (DOA'07), Vilamoura, Algarve, Portugal*, 2007.

[20] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A Framework for Rapid Integration of Presentation Components. In *Proceedings of the International Conference on World Wide Web (WWW'07), Banff, Alberta, Canada*, pages 923–932, 2007.