DIA: DATA INTEGRATION USING AGENTS

Philip S. Medcraft, Ulrich Schiel, Cláudio S. Baptista
Departamento de Sistemas e Computação, Universidade Federal de Campina Grande
Av. Aprígio Veloso, 882, Campina Grande, PB Brazil
Email: philip@dsc.ufcg.edu.br, ulrich@dsc.ufcg.edu.br, baptista@dsc.ufcg.edu.br

Keywords: Mobile agent, heterogeneous databases, data integration, ontology, semantic web, xml.

Abstract: The classic problem of information integration has been addressed for a long time. The Semantic Web project is aiming to define an infrastructure that enables machine understanding. This is a vision that tackles the problem of semantic heterogeneity by using ontologies for information sharing. Agents have an important role in this infrastructure. In this paper we present a new solution, known as DIA (Data Integration using Agents), for semantic integration using mobile agents and ontologies.

1 INTRODUCTION

The classic problem of information integration has been addressed for a long time. One of the main goals of an integrated system is to enable a query environment in which users tell what they are looking for without knowing how and where the information is stored. Then, there are at least two approaches for data integration:

- Materialized: data are extracted previously from the data sources and integrated in a unique database, from which users can query this data;
- Virtual: data are extracted on the fly from the data sources.

Obviously, there is a trade-off between these previous approaches. In the materialized one there is no schema translation on the fly, however it is more difficult to maintain the warehouse up-to-dated. On the other hand, the virtual approach always obtains up-to-dated data, however it is necessary schema translation on the fly.

Three database architectures have been proposed to solve the data integration problem: Federated databases, which enforces the virtual approach; data warehousing, which is based on the materialized approach; and mediators, which uses the virtual approach.

Garcia-Molina (Garcia-Molina et al, 2002) discusses some of the important issues regarding data integration:

- There are different data types in the different sources;
- Data sources may use different names or format to refer the same object;
- Terms may have different semantics in the way they are stored in the data source;
- There are data that are not present in all data sources.

The Semantic Web project is aiming to define an infrastructure that enables machine understanding of data on the Web. This is a vision that tackles the problem of semantic heterogeneity by using ontologies for information sharing. Figure 1 (Berners-Lee, 2000) presents the Semantic Web vision. Although designed to be used on the entire Web, the ideas behind the Semantic Web can be used in a federated database environment, in which the data sources are previously known and there is a global schema, described in an ontology, enabling data integration.

The three database architectures previously mentioned undertake a considerable amount of data flow and the unnecessary visiting of data sources.
Mobile agents represent an excellent option to solve these problems.

Figure 1: The Semantic Web Layers

In this paper we present a new solution, known as DIA (Data Integration using Agents), for semantic integration of a federated database using mobile agents and ontologies. The availability of an explicit ontology for the data sources is at the core of the integration process, as it represents the global schema of the federated database. The rest of the paper is outlined as follows. Section 2 describes the DIA architecture. Section 3 addresses design and implementation issues. Section 4 discusses related work and presents our main contributions. Finally, section 5 concludes the paper and discusses further work to be undertaken.

2 DIA ARCHITECTURE

Traditional information retrieval over the Internet employs the client-server architecture. The exchange of direct communication between the information request site and the information sources by mobile agents can reduce the overhead of the communication link, used to transmit the client requests and the server replies, by moving the knowledge about the retrieval close to where the information resides.

The support for disconnected operations and the efficient execution of global queries are some of the many advantages in using mobile agents for this kind of application. The latter can be greatly improved by having an ontology defining the global concepts and how they interconnect semantically with the schema in the distributed database.

Figure 1 (Berners-Lee, 2000) shows the proposed layers of the Semantic Web with higher-level languages using the syntax and semantics of lower-level languages. This project focuses on the ontology language level and the agent-based computing enabled by ontology languages. Higher-level languages (with complex logics and the exchange of proofs) will enable even more interesting functionalities, but we’ve left those to be discussed in future projects.

The objective of our architecture is to provide a uniform interface to heterogeneous, pre-existing RDBMSs, in a way that users can formulate queries (global queries) that are transported between these different databases by a mobile agent to produce a single consolidated response. For this purpose, it is necessary to identify the objects in the databases that represent equivalent or similar concepts, that is, are semantically related. Heterogeneities between the databases appear as semantic conflicts. They are detected and solved at the moment a new database enters the federation.

Once identified the equivalent or similar concepts in the new database, they are defined as global concepts in the existing ontology. This ontology describes the concepts as classes and properties, and defines the relationships between each other. The uniform interface of the user to the databases should be based upon this ontology so that a global query is constructed with terms that relate to concepts present in the global database schema. Therefore, the query adapting process on each local database will occur by verifying the relation between each global term and its occurrence in the local database schema. By applying this mechanism, we solve the semantic conflicts in each local database.

As we described previously, the use of an ontology provides the required semantics for heterogeneous conflict resolution. Now we will describe how our mobile agent architecture can improve the process of retrieving and consolidating data from the heterogeneous database.

There are several design patterns for agent-based applications. These patterns are usually divided into three classes: traveling, task, and interaction patterns (Lange & Oshima, 1998). In our architecture we propose the use of a traveling pattern and a task pattern only, but interaction patterns are also perfectly applicable in our context.

Traveling patterns deal with various aspects of managing the movement of mobile agents. These patterns allow us to enforce encapsulation of mobility management, which enhances reuse and simplifies mobile agent design.

In our architecture, we propose the use of an extended Itinerary traveling pattern. A basic itinerary pattern maintains a list of destinations and always knows where to go next. It also defines special cases such as what to do if a destination is temporally unavailable. In the extended version the
The itinerary can be changed dynamically, i.e., the mobile agent is able to decide where to go depending on the result of its local task.

To enable a host to be part of the mobile agent destination list, every destination needs to have an agency running a local stationary agent, which cooperates with the mobile agent in the local query adaptation process.

Task patterns are concerned with the breakdown of tasks and how these tasks are delegated to one or more agents. In our architecture, we propose the use of the Master-Slave pattern, which allows a master agent to delegate a task to a slave agent. The slave agent moves along its itinerary, performs the specified task and returns the result to its master agent.

As shown in figure 2, the user triggers the process by preparing a global query, based on terms from an ontology. When the user submits the query, the application notifies a stationary agent running inside an agency. This stationary agent (master agent) creates a mobile agent (slave agent) for the migration through the databases. Applying the Itinerary pattern explained previously, before its creation, a list of destinations is set for the new mobile agent (see section 2.2). With the query in hand and knowing its itinerary, the migration process starts.

As the mobile agent reaches a destination, it contacts the local stationary agent, passing it the global query. This agent knows the ontology on which the global query terms are based on, and also knows the local database schema. Therefore, with a local matching mechanism, it is able to convert the global query to a corresponding local one. The database query is performed and the result set presented to the visiting mobile agent in XML syntax.

In case the mobile agent is not at the first destination, before moving to the next node, the mobile agent performs an integration of the local result with the result collected previously (see section 2.3). This reduces the size of the carried XML, and means that when the mobile agent returns to its origin, no extra result integration or analysis will be performed.

When the mobile agent completes its itinerary (see section 2.4), it migrates back to its origin, returns the final result to its master agent and removes itself. The master agent presents the collected data to the user.

[Diagram of architecture for database integration]
2.1 DIA query interface

The interface was developed for web browsers. A tree is automatically built from the DAML+OIL ontology classes. Moreover, the fields for the input query are also built according to the ontology properties for each class.

Figure 3 shows an example of the interface. The tree on the left side presents three classes: Person, Employee and Customer. The classes Employee and Customer are both subclasses of Person. In the example, the user has selected the class Customer. Therefore, its properties are shown on the right side of the interface, organized in a HTML form for the input query definition.

The query defined in figure 3 means that we want the sum of all credits obtained by a specific customer, and has the following SQL equivalent:

```
SELECT CPF, SUM(CREDITS)
FROM Customer
GROUP BY CPF
HAVING CPF = '027571664-33'
```

2.2 Preparing the mobile agent itinerary

In our architecture, all nodes participating on the corporate database need to have an agency running a local stationary agent. This stationary agent is responsible for the global query adaptation, for retrieving data from the local database and for returning the retrieved data to the visiting mobile agent.

The master agent (who creates the mobile agent) knows all the local database schemas, and is able to discover which local databases can have data that attend at least partially to the global query. Only the hosts that shelter these databases should be included in the mobile agent itinerary. Doing so, the mobile agent does not visit hosts unnecessarily.

2.3 Local integration

After the global query adaptation and having retrieved data from the local database, the local stationary agent presents the retrieved data to the
mobile agent in XML syntax. Figure 4 shows an example of how a tuple with the attributes of the query defined in section 2.1, retrieved from the local databases, should be presented to the mobile agent.

```
<row version="1.0">
  <name>...</name>
  <operation>...</operation>
</row>
```

Figure 4: Result example in XML syntax

The element “column” has two attributes: “name” and “operation”. The former presents the result set column names, and the latter presents the aggregation operation to be performed on each column during the integration process. To make possible the integration process, the column names are based on the global query terms.

The following operations can be used:
- undefined: maintain the existing value;
- sum: add the value of column to the existing value;
- min: take the minimum of the two;
- max: take the maximum of the two;
- avg: take the average of the two.

The local query adaptation should guarantee that each resulting row has a column that identifies it uniquely in the result set. The integration of a current XML with a new XML occurs by comparing their rows, performing the following steps:
- If a row key column in the second XML matches a row key column in the first XML, all the other columns are integrated based upon their operations;
- If a row key column in the second XML does not match a key column of any row in the first XML, the current row is appended to the first XML;
- If a row key column in the second XML matches a row key column in the first XML, but the second XML has a column that does not exist in the first XML, then the column is appended to the current row in the first XML;
- The “undefined” operation is set for every non-numeric attribute. If a conflict occurs the system maintains the current value.

Figure 5 shows an integration example.

```
<row version="1.0">
  <name>...</name>
  <operation>...</operation>
</row>
```

Figure 5: Integrating two XML results

This local integration policy guarantees that a minimum of necessary data is moved from one site to the other and, at the end of the itinerary, the result set is ready to be presented to the user.

2.4 Improving the Itinerary pattern

Since we want to obtain the best itinerary for a given global query, several points are considered in order to avoid the visit of unnecessary hosts. First we distinguish between static and dynamic itineraries.

A static itinerary is an itinerary that can be established in advance by the stationary master agent. Depending on information available about the local databases, the master agent analyses the global query and selects those local databases that must be visited to process the query. In the simplest case, the itinerary is “visit all nodes”.

In a dynamic itinerary, the mobile agent can decide, depending on the partial result he gets, if the query is already answered and he can return, or not. This dynamic capability depends on a previous classification of the global query. For instance, queries as “What is Philip’s birthday?” or “Are Philip’s credits more than 700?” allow a dynamic itinerary, whereas “Give me Philip’s total credits” needs always a static itinerary.

Therefore, the constructions of the itinerary (full or selected) and its classification (static or dynamic) is done by the master agent analyzing the SQL structure of the global query together with the local schemas.

3 DESIGN AND IMPLEMENTATION ISSUES

The system analysis and design were based upon a methodology for agent-based applications proposed by (Guedes, 2001), producing UML
artifacts (Unified Modeling Language) (Rumbaugh et al., 1999; Jacobson et al., 1999). The system was developed using the JAVA and Grasshopper Platforms (IKV++, 2002). The former was used to construct the internal classes and interfaces of the system, and the latter was used as the system agent platform. We have also used some design patterns (Iterator, Observer, etc.) to improve the system design quality (Gam et al., 1999).

The class diagram of the system is shown in figure 6. We emphasize the **RelationalWrapper** class, which is responsible for the query local adaptation process. This class has three methods: the **matchSchema()** method translates the query terms to the appropriate terms of the local source; **prepareSQL()** structures the query before retrieving data from the local database; **processXML_SQL()** structures the query answer in XML syntax.

The Grasshopper agent platform (IKV++, 2002) permits local communication as well as remote communication between its components (agents, agencies and regions) and external objects.

A desired property of mobile agent systems is localization transparency. In other words, the entity that wants to communicate with a mobile agent does not need to worry about its localization. To be able to provide this characteristic the Grasshopper platform uses a communication pattern called Proxy. Proxy consists of an intermediary entity on the communication between a client and a server, being responsible for the connection establishment and for identifying the server localization. The class diagram on figure 6 presents three agent classes: **MasterAgent**, **MobileAgent** and **LocalBDAgent**. As shown in the figure, each class is associated to a correspondent proxy class, which will intermediate the communication to any external objects (agencies, other agents or external objects).

The **MasterAgent** is a stationary agent and is responsible for listening to the user. When the user
submits a new query, this agent sets up the itinerary, creates the *MobileAgent* (slave agent), and delegates the realization task to it. The *MobileAgent* then migrates through its *Itinerary* and as it reaches each destination, it contacts the *LocalBDAgent* that is responsible for adapting the query locally, for retrieving data from the database, and for returning the local result to the *MobileAgent*. Then, it integrates the results and continues its itinerary.

### 4 RELATED WORK

The literature lacks works on data integration using an ontological approach plus a mobile agent approach. In this section we discuss research works related to our approach.

- **Observer (Ontology Based System Enhanced with Relationships for Vocabulary hEterogeneity):** This approach for query processing in a Global Information System (GIS) uses multiple pre-existing ontologies, related to each other or not, in the underlying data repositories. One or more ontologies describe each repository using Description Logics (DLs), which are then translated to the local query languages of the data repositories (Mena et al, 1996).

- **Ontobroker:** This project uses ontology, based on Frame Logic, and deductive inference systems to provide access to semi-structured data (Erdmann & Studer, 1999).

- **MEDIWeb (A Mediator-Based Environment for Data Integration on the Web):** This approach presents an architecture of a web-based query system in which users, by using an ontology, can specify their queries and submit to the underlying data sources. These data sources can be either database systems or XML files (Arruda et al, 2002).

- **Integration of Databases using the Mobility of the Code:** this approach uses code mobility to obtain information retrieval in heterogeneous databases (Claro & Sobral, 2001).

When comparing these approaches with our approach, we can argue that only the latest considers using mobile agents. However, it does not use an ontology approach and therefore lacks the agent-based computing ontologies enable. Moreover, it lacks code portability. DIA realizes an optimized itinerary and performs a sophisticated integration of its results. Furthermore, in our system, every class is portable. An ontology-schema matching table is all that is necessary to add a new database to the integrated environment.

The related approaches lack the mobile agent advantages, like: reduction in network load, dynamic adaptation, reduced communication costs, autonomous execution, and network latency overcoming.

### 5 CONCLUSION

In contrast to free web search approaches, where the universe of searched sites is highly dynamic and unknown to the client machine, there are a lot of highly complex applications, such as world corporations, governments, and multiple shops with a distinct environment. They have a well-known set of databases with information possibly highly related together. In such an environment, known as Federated Database System or Multidatabase System, the knowledge of the information stored and the relative stability of the members of the federation can be used to improve the processing of global queries.

We have presented a solution for this situation, which uses a global ontology to inform the user what information is stored in the whole enterprise and helps the system where to get the results for a specific query. The processing of the query is realized by a mobile agent, which gets its job from a global stationary agent and runs the federation in order to request from local stationary agents the information needed. The result sets are joined pair wise and after each local processing the mobile agent decides if it is necessary to continue or he can return to the host. This circular letter strategy seems to be better than other approaches, such as mediator based ones, since it relieves network communication and its sensibility to the real information needs for a particular query, reduces unnecessary data flow.

Among the selection of the nodes of an itinerary and its classification as static or dynamic, the order of the itinerary traversal could be significant. If the more “light” nodes are visited first, the amount of data the mobile agent takes along could be reduced. On the other hand, in a dynamic itinerary, the order could determine an earlier return to the origin agency. This ordering of the itinerary can only be performed if the origin agency has statistical information about the local databases.
REFERENCES


DAML+OIL 2001: Available at: http://www.daml.org/2001/03/daml+oil-index

