1992

Integrating Distributed Data Sources Using Federated Objects

Henry R. Tirri

Jagannathan Srinivasan

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
92-023

INTEGRATING DISTRIBUTED DATA SOURCES USING FEDERATED OBJECTS

Henry R. Tirri
Jagannathan Srinivasan
Bharat Bhargava

CSD-TR-92-023
April 1992

Integrating Distributed Data Sources Using Federated Objects*

Henry R. Tirri†
Jagannathan Srinivasan
Bharat Bhargava
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

We address the problem of integrating heterogeneous data sources for collaborative environments such as Groupware systems. Pre-existing related distributed data is modeled by a *composite* object, where each data source is treated as a fragment of the composite object. We define a collection of methods which allow manipulation of the related distributed data in a controlled and consistent manner. The composite object created in this manner is referred to as *federated object*, and it provides a pragmatic approach to data integration. Our approach is different from the approach of integrating heterogeneous database systems, as we do not require each data source to have full database system capabilities. Instead, for the collaborative application environments the data sources can be simple files, application generated binary files, or commercial data servers. The federated objects are especially suitable for computer-supported-cooperative work (CSCW), where coordination and communication among a group of people is required, and the type of collaboration is dynamic in nature. In this paper, we discuss the general properties of federated objects, present implementation principles for the required mappings, and show how federated objects can be used to implement a simple report writer application that needs to access several distributed data sources. A fully developed federated object mechanism relies on a "toolkit" approach, where federated objects are constructed from existing data sources using a C++ class library, which provides the mechanisms needed for access and manipulation of a wide variety of data sources.

*This research is supported in part by a grant from AIRMICS and UNISYS.

†On leave from Department of Computer Science, University of Helsinki, Finland.

1 Introduction

Over the past five years, attention to collaborative computing has increased dramatically. Collaborative computing uses networking, communications, concurrent processing, and windowing environments. Colab project [FT88] introduced the notion of seamlessness between individual and group work, that is, the merging of group tools with individual tools. However, developing a self-contained and wholly integrated environment over a variety of computer systems that support both individual and cooperative work, has proven to be a difficult task. An important issue to be addressed is how existing (individually owned) data sources can be integrated without violating their normal use, i.e., individually owned data can be shared among several cooperating users in a flexible manner.

Several approaches have been used for integrating existing data to allow access and updates in a controlled and consistent manner. The research efforts in building federated database systems [SL90, ASD+91, Gup89] focus on integrating heterogeneous database systems by providing a software layer on top of the database systems. However, these attempts assume existence of a full capability database system at each of the sites. In the collaborative environment there are many cases where data is managed in an ad-hoc manner and only minimal services are provided to the user. For example, data may exist simply in files and a collection of programs may be used to manipulate the data, in which case the set of possible operations are "hardwired" into the individual programs.

To deal with data having a wide variety of access capabilities, ranging from a minimal access (say through a collection of programs) to access via a full database system with support for transactions, we propose an object-oriented approach for *loose* integration through special "data integration" objects. Our approach is similar to the approach adopted by Distributed Object Management Systems such as DOM [BOH+91]. However, our main goal is *data integration*, not the general integration of applications and our target environment is characterized with frequently changing collaborative needs that prohibit development of complex mechanisms. Also in our case, providing a high degree of autonomy in manipulation of individual data sources is more a rule than an exception; the owners of individual data sources are willing to share their data as long as it does not interfere with their primary activities.

We propose integrating related distributed heterogeneous data by a *composite object* [KBG89], where the data residing on each site is viewed as a *fragment* of the composite object. A set of *methods* is defined to manipulate these fragments collectively as a single entity in a consistent and controlled manner. Such a *fragmented composite object*, referred to as *federated* object, allows loose integration of data residing on multiple sites. The federated object is created in a *bottom-up* fashion as opposed to *top-down* creation of composite objects in distributed homogeneous database systems [BBS92].

Data integration through federated objects is aimed for collaborative applications, i.e., Groupware, which require *dynamic grouping* of data according to individual user's (or group's)

needs. On the other hand, owing to the nature of collaboration, the original data sources can still be accessed and manipulated by the individual users, and an essential requirement of such an integration environment is that the individual user should not be disturbed by the sharing mechanism as too much interference would reduce the willingness of the individual to participate in the collaborative activity.

An example of such a collaborative activity is preparing a project report that involves access to several independently manipulated data sources such as text files, files containing figures in various formats, speadsheet generated files containing budget information, etc. In each of these cases, the individual users should be able to access the data sources they own directly even after the integration, without being affected by the collaboration level activities. For example, accounting users should not be hindered in entering new cost information to the speadsheet files just because the editor of the project document has already placed an earlier version of the data in the document. Our federated object approach allows application programmers to use various data sources without requiring the original individual users to conform to the use of the new group application, or its interface, a fact that reduces the resistance to share data in a collaborative environment.

In [TSB91] we used the notion of fragmented composite object to model distributed data in a database environment. We proposed that for distributed applications such as airline reservation systems the resources (airline tickets) should be distributed to multiple sites allowing independent processing at each of the sites. We developed an efficient transaction mechanism for such applications. However, in such an environment the composite objects are created in a *top-down* manner and the process resembles fragmentation schemes used in relational database systems. We are also investigating the replication of fragments of composite objects in a distributed homogeneous database system [BBS92]. A distributed object-oriented database system called O-Raid [DVB89, BDMS90, MSDB90] is being used to experimentally evaluate the benefit of selectively replicating fragments. In this approach we assume that the fragment dependencies are captured implicitly via methods.

Here we adopt a pragmatic approach for integrating data sources. This is based on the observation that the existing heterogeneity is not only due to the various representation formats, but is also due to the different operational capabilities supported by the sources. We adopt a "bottom-up" strategy, that is, we assume the existence of data sources with certain representation formats (e.g., ascii text, postscript, or binary files) as well as standard capabilities (e.g., an SQL interface) which cannot be changed. Our federated object mechanism encapsulates such dimensions of heterogeneity to allow flexible application programming. We are especially interested in environments, where the integration needs are dynamic: the set of data sources involved in a collaboration changes frequently. An example of such a collaborative activity is a short-term project documentation, at the end of which the integration of the various "source parts" of the report becomes obsolete. This is in contrast to the traditional database integration (homogeneous or heterogeneous), where the configuration of the set of participating databases is usually stable.

In Section 2 we outline the federated object architecture. Section 3 presents how federated objects can be manipulated as well as the discusses interesting properties of such objects. Section 4 describes the implementation of a simple report writer application using the notion of federated object. Finally, we give conclusions and outline the future work planned.

2 Federated Object System Architecture

Layered system architectures result in clean and open systems. Unfortunately having many layers imposes a heavy penalty on the system performance. For practical purposes, the federated object system architecture has only two layers: the Data Source layer and the Federated layer. As the primary goal is to provide a tool for short-term data integration (e.g., for a temporary cooperative work in preparing a report), the complex support beyond these two levels is usually not required. Thus, the system does not support hierarchical federated objects. The federated object system architecture is illustrated in Figure 1. We identify the following components:

- one or more data sources that allow external data access (Data Source Layer),
- one or more Federated Objects (FO[1]) that can share data sources via fragments (Federated Layer), and
- mappings between the FO fragments and the data sources.

In general, the data sources (files, databases, etc.) can be accessed independently by other users without requiring all the accesses to be performed through the FO-interface. This is typical of applications where the owners of the data source are willing to allow access to their data as long as it does not affect their primary activities. Later we discuss the enhancements to this scheme, where users by accepting the use of a given filter (tightly-coupled cooperation) can improve the performance characteristics of the federated access.

2.1 Data Sources and Access Capabilities

Federated object allows integration of data sources with a wide range of access capabilities. In principle, these data sources can be classified into two categories:

1. *Passive data sources.* Ordinary files with read/write (blind write) capabilities fall into this category. A limited set of operations are supported, and the data sources lack communication support. Examples of such passive data sources are ascii text files, program generated binary files, etc. For such data sources only read and/or write

[1]The abbreviation FO is also used in [MNS92] to denote fragmented (active) object. However, these fragmented objects model distributed applications.

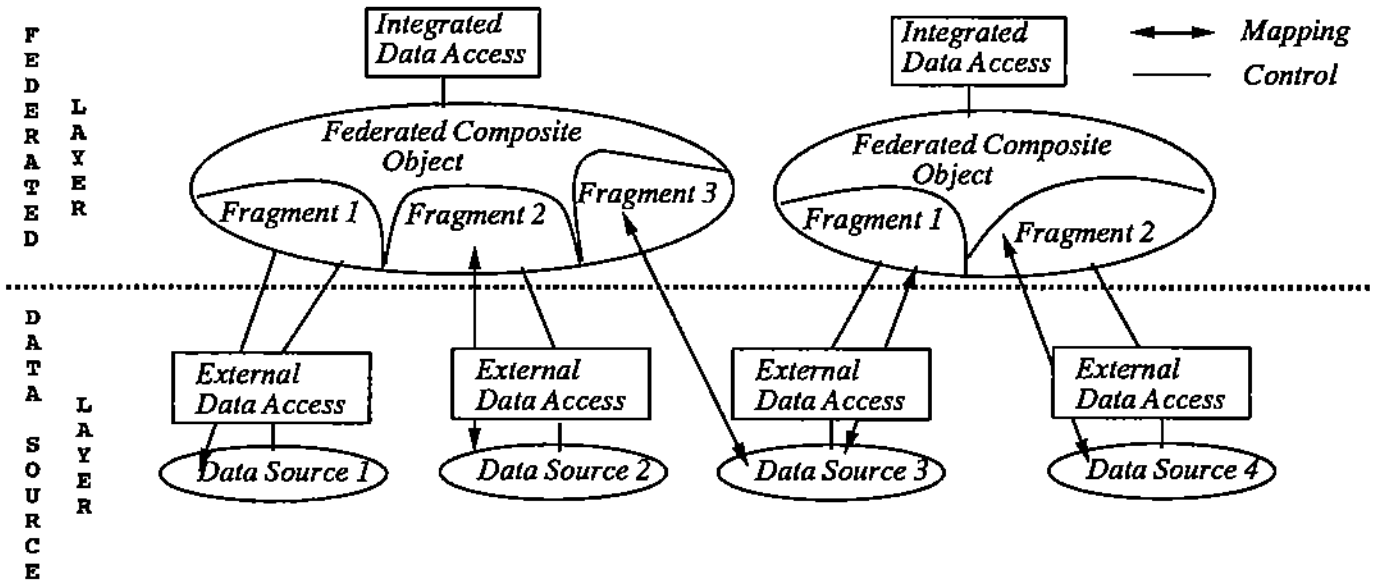

Figure 1: Federated Object Architecture

operations are allowed, and all the processing support is handled at the federated level. These operations can involve the entire file or a selected part of the file.

2. *Data servers.* This category contains data sources with a range of access capabilities as provided by the corresponding data server. On one end, the data server can provide a query language interface to access and manipulate data stored in a database. Note that the data source need not encompass the entire database. For example, in relational database systems, a data source can be one or several relations. Also on the other end, the data server may provide only read access to the data it manages.

For a meaningful integration, at least access capability must be allowed on the data sources, whereas the update capability is optional. Sometimes, only the update capabilities are meaningful, but for simplicity we assume that for those data sources we also have access capabilities.

2.2 Federated Object Structure

A federated object is composed from the following type of data members (see Figure 2):

- *Fragment.* One fragment per data source is created. The fragment can be either *embedded* in the federated object or can *reside elsewhere.*

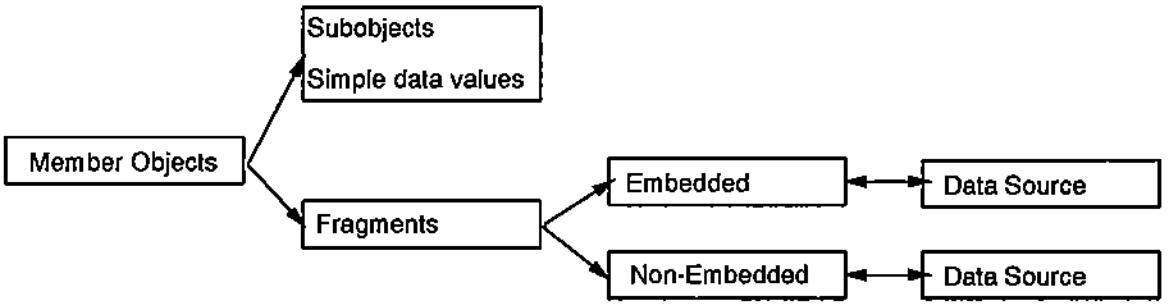

Figure 2: Components of a Federated Object

- Embedded fragments improve performance as the translated data is maintained as part of the federated object. However, mutual consistency between embedded fragment and corresponding data source has to be maintained. The decision to use embedded fragments depends on the expected access frequency for the data source as well as the consistency requirements. For example, if a data source is frequently accessed and rarely updated, having an embedded fragment will reduce the communication and translation overhead. Similarly, if the application can work with older versions of the data even when there are external (individual user) updates, the embedded fragments will be useful. For example, older versions of a customer data can be used in the design phase without sacrificing a meaningful end result.
- Non-embedded fragments are useful for the data sources that are infrequently accessed. Also, the non-embedded fragments allow interfacing with database systems. Often, an interface is needed for the entire database and the data to be accessed is not known in advance.

  Note that the retrieved data can be cached even for non-embedded objects (similar to caching of results of procedure attributes in POSTGRES [RS87]). However, for clarity of the presentation we do not consider supporting caching of non-embedded fragments.

- *Subobject and simple data value.* These data member contains data common to one or more data sources. For example, the subobject may contain *aggregate* values generated from one or more fragment values.

The federated object constructed from different data sources are accessed and manipulated by a collection of *methods*. The methods can access more than one fragment of the federated object and hence can be used to capture *implicit* dependencies between different data sources.

2.3 Mappings

For each data source two translation routines need to be defined between the data source and the corresponding federated object fragment:

1. `fragment_to_data` routine maps federated object fragment to the data source.

2. `data_to_fragment` routine maps a data source to corresponding fragment.

These translation routines are defined as methods for the federated objects. Unlike other methods, these methods can update at most one fragment. This keeps the mechanisms simple and avoids the complex issues raised by the distributed updates. However, the method can access and update any number of subobjects and data values. Naturally the reverse mapping `data_to_fragment` need not to be defined if the corresponding data source is only accessed and never updated.

3 Manipulating Federated Objects

The creation of a federated object itself is done through the execution of constructor methods. Additional methods can be defined for accessing and manipulating the federated object. In the following we illustrate the capabilities of such methods and how they should be designed.

3.1 Restricting mappings for creating Federated Objects

As discussed earlier, for each data source a data_to_fragment translation method has to be defined. In our case such a translation method can update exactly one fragment object, although there is no restriction on the subobjects and data values it updates. In addition we restrict the translation method not to be able to access (read) any part of the federated object. The mapping from data to the fragment performs a *blind write* on the federated object in question. This restriction on mappings simplifies the creation of federated object as different fragments can be generated from the data sources in an arbitrary order. Specifically, let m_i and m_j be two translate methods that generate fragments from data source i and j respectively. The federated object FO will reach the same state for the two execution sequences, namely, m_i followed by m_j and m_j followed by m_i. In other words, the execution of translation methods is commutative.

3.2 Accessing Federated Objects via methods

The methods defined for federated object can be classified as follows:

- Methods that read parts of the federated object.

- Methods that update parts of the federated object.
- Methods that read as well as update parts of the federated object.

The advantage of using the first two types of methods is that the required system overhead is much less than for the last type. Thus the last type of methods should not be used unless they are really needed.

Methods accessing parts of federated objects For pure read methods there are two cases:

- The methods access fragments only, not subobjects or common data values. It is sufficient to ensure that the accessed fragments are up to date. This check can be done by maintaining a bit vector, with one bit per data source. The bit is set whenever a data source has been updated, but the corresponding update is not propagated to the federated object at this time. The propagation is delayed until the fragment is really accessed. At the access time, simple *and* operation on bit vector can be employed to determine if the fragment being accessed is out of date with respect to the corresponding data source. If so, the corresponding constructor method is executed to regenerate the fragment and update the common subobjects and data values. The advantage of this approach is that if a data source is updated several times between accesses, only one propagation effort is needed. This reduces both communication cost and blocking delays.
- The method access the common subobjects and/or data values. In the general case, we do not know the data sources on which the subobject being accessed depends. In such a case the bit vector containing data source up to date information is examined. If it is non-zero, that is at least one data source has more recent data, the constructor methods are executed for corresponding data sources to bring the federated object to an up to date state.

The above strict scheme will result in bringing the entire federated object up to date before accessing the common subobjects and data values. Often, it may be enough to allow a demand driven update mechanism where a method accessing common subobjects and data values can avoid bringing the federated object up to date, especially if the subobject and data values accessed do not depend on the underlying data source that has changed. To support such a scheme the user is required to define the dependency of the method on the corresponding data sources.

For example, if a method m depends only on data sources ds_1 and ds_3, that is,

$$m : ds_1, ds_3$$

then prior to execution of the method m, only the updates corresponding to data sources ds_1 and ds_3 need to propagated. Such explicit dependency specifications give additional flexibility to our system, if such dependencies are not known or they are too difficult to be analyzed, the above strict scheme can be followed.

Method updating fragments Due to the natural semantics of federated objects the members of such an object are related. There usually are dependencies between a subobject and a fragment. Updating a fragment may involve updating one or more subobjects. Such dependencies can be *implicitly* captured in method definitions. Since a fragment is always updated by a method, the method can be defined to update other related subobjects to bring the entire federated object into a consistent state. Note that in general there could also be dependencies between two fragments. Such dependencies are harder to capture; especially creating fragment from the data source can no longer be done independently in arbitrary order.

Method updating fragments can cause inconsistency as the equivalent updates have to be propagated to corresponding data sources. Since the data sources are autonomous and can be independently updated, in general the update propagation may not always succeed. The problem of providing multi-fragment update methods is in general as difficult as the update problem in heterogeneous multidatabases [SL90]. Thus we only support methods that update at most one fragment.

3.3 Properties of Federated Objects

Embedded Fragment Mutual Consistency If a fragment is embedded in a federated object, then mutual consistency between the fragment and corresponding data source needs to be maintained. The mechanism has to address two types of updates:

- *Updates on Embedded fragment*: The update has to be propagated to corresponding data source. For simple data source such as a file the data source is overwritten (*blind-write*) after generating the equivalent file using the `fragment_to_data` method. If the underlying data source is controlled by a data server, then an appropriate update request is sent e.g., in SQL.

- *Updates on Data Source*: Similarly updates have to be propagated to the corresponding fragment. Two schemes are possible:

    1. *Instantaneous Update*: The user performing the update invokes an additional method (defined for federated object), which regenerates the fragment and any other associated subobjects and data values.
    2. *Demand Driven Update*: The update on the data source only causes updating the timestamp associated with the data source. Almost all types of data source

have such timestamp information indicating when they were last updated (e.g, file header information, database version information). Now when the corresponding fragment is accessed first time after the data source update, the timestamp of fragment is compared with that of the data source. If there is a mismatch, a method is invoked to regenerate the fragment and related subobjects and data values.

Non-embedded fragments The non-embedded fragments can be implemented through methods that retrieve data from the corresponding data source. For example, a method could simply be a query in a relational database system. Here there are no consistency problems as a fragment is derived on demand and is automatically up to date. This corresponds to the view mechanism in databases [EN89]. Updates on non-embedded fragments are done by submitting an appropriate update request to the corresponding data server. For improving performance, the data retrieved through the method can be cached. In that case, one has to deal with the problem of cache getting out of date due to updates on the underlying data source. Such cache consistency issues are clearly related to the notion of quasi-copies and many of the mechanisms discussed in [ABGM90] can be applied.

Atomic Execution of Methods When a method updates a fragment the update needs to be propagated to the corresponding data source. If a method manipulates more that one fragment, the atomic execution of method is more complex. Specifically there are two cases:

- All the manipulated fragments are embedded. In this case the updates can be performed. However, the equivalent updates on the corresponding data sources need to be propagated.
- One or more fragments are pointers, i.e., data sources are either servers or nonresident files. In this case the updates on the corresponding data sources can fail. For such a case a set of methods that implement transaction support (e.g., commitment, isolation) is needed. For an example of such a mechanism see [TSB91].

4 Building Applications using Federated Objects

Applications can be built using the notion of federated objects. To reduce the effort involved in building applications, a collection of C++ [MS90] classes (system classes) is provided. These classes support the basic mechanisms required for the access and manipulation of various (standard) types of data sources. In this section, we describe these systems classes, outline the steps involved in building an application, and present the process layout of the application built in this manner.

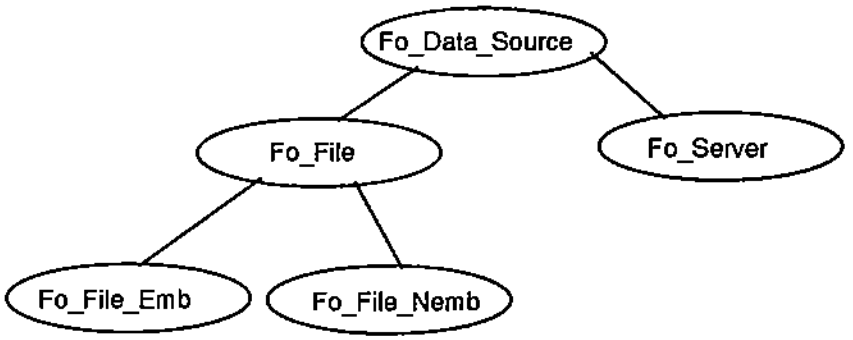

Figure 3: Federated Object System Class Hierarchy

4.1 Federated Object System Classes

For each type of data source a separate class is defined. All federated object system classes are prefixed with `Fo` to distinguish them from the user-defined classes. Figure 3 shows the system class inheritance hierarchy. All classes inherit from the system class `Fo_Data_Source`. This class contains attributes and methods common to all data sources as shown below. For example, the method `get_timestamp` obtains the timestamp of the last update performed on the data source. The timestamp information is used to determine if a fragment data is out of date.

```
class Fo_Data_Source{
char *name;                    //uniquely identifies the source
char *site;                    // host site name
long timestamp;                // when the data was last modified
virtual long get_timestamp(); //obtains timestamp of last update
};
```

The method code for the system classes is divided into two files:

1. *Client code*: The client code contains the method stubs which are linked with the application program.

2. *Server code*: The server code contains the detailed implementation of the methods. These methods are invoked via Sun[2] Remote Procedure Call (RPC) [Sun88] mechanism from the application.

[2]Sun is a registered trademark of Sun Microsystems, Incorporated.

4.2 Writing an Application

The applications are written in C++ programming language. The application writer composes a federated object from the desired data sources. The steps involved are as follows:

1. Identify the set of data sources that need to be integrated.
2. Define the federated object structure:
   (a) Define subobjects and data values
   (b) For each data source define the fragment structure. If the data source is of type `Fo_File` then this involves deciding if the fragment will be embedded or non-embedded and respective structure definition. If the data source is of type `Fo_Server` then by default the fragment is of non-embedded type.
3. Define two translation methods per data source. If no translation is required then the *identity* function is used as the translation method.
4. Define a set of methods to manipulate the federated object. These methods can potentially invoke any of the methods defined in the classes of the included data sources.

Example: A simple Report-Writer application To illustrate the design, let us consider a *Report-Writer* application. The application uses multiple data sources (files) to form a report. We built the application using X [SG83] windows. Figure 4 shows the Report-Writer with access to two data sources. The data source is specified by giving a pair: (site, pathname). Each of the included data sources is displayed in a separate window. User can perform read and write operations by clicking the *Read* and *Write* buttons. The read and write operations (methods) apply on individual data sources. In addition, there are two methods *Word-Count* and *Spell-Check*, which apply collectively to all the data sources. Clicking on the Word-Count displays a count of lines, words and characters (similar to `wc` command in UNIX[3]) of the entire report. Clicking on the Spell-Check displays spelling errors in the entire report (similar to `spell` command in UNIX). All messages are displayed in the bottom *Execution Trace* window.

The application uses `Fo_File` system class, and is compiled and linked with the client module for the `Fo_File` system class.

4.3 Process Layout

Invoking the application results in creation of one *application process* and several *server processes* (one per data source). For example, the execution of Report-Writer results in three

[3]UNIX is a trademark of AT&T Bell Laboratories.

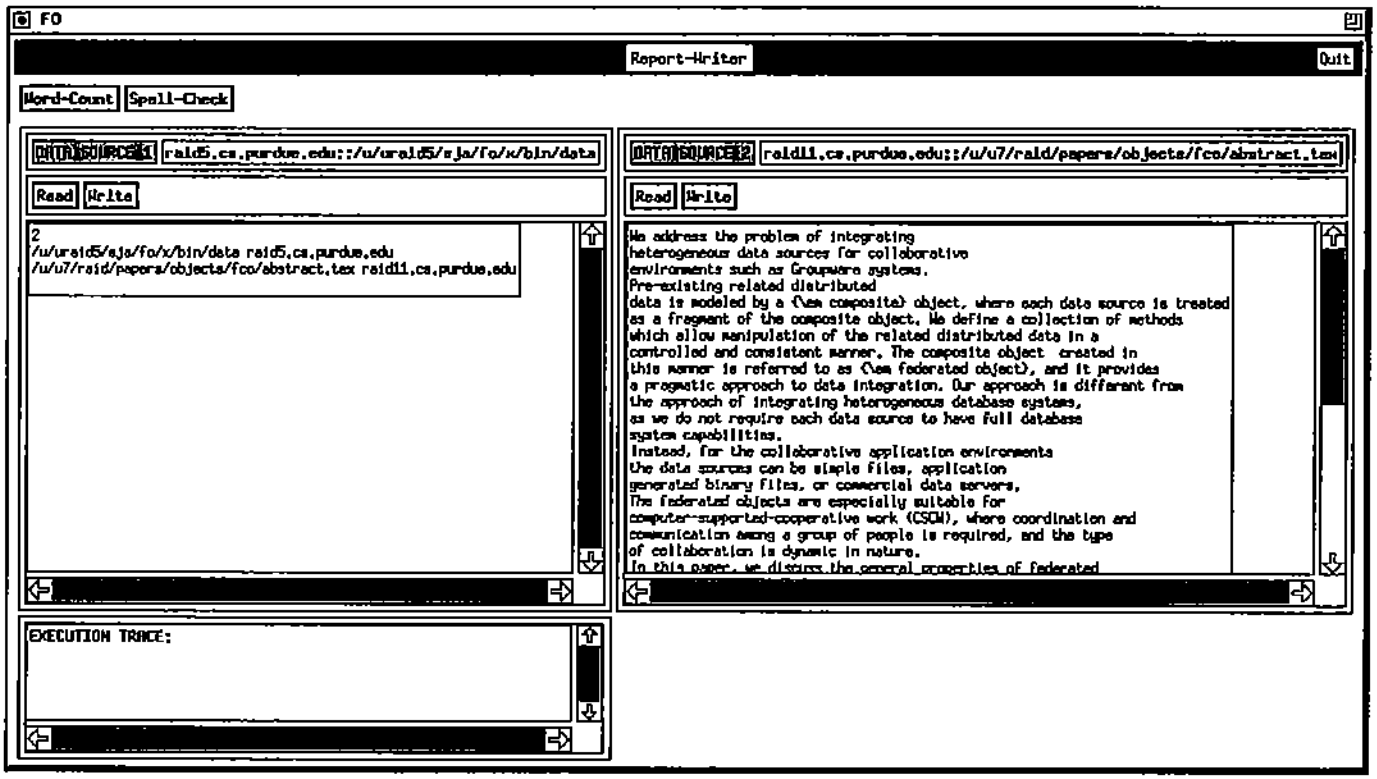

Figure 4: The Report-Writer application with two data sources

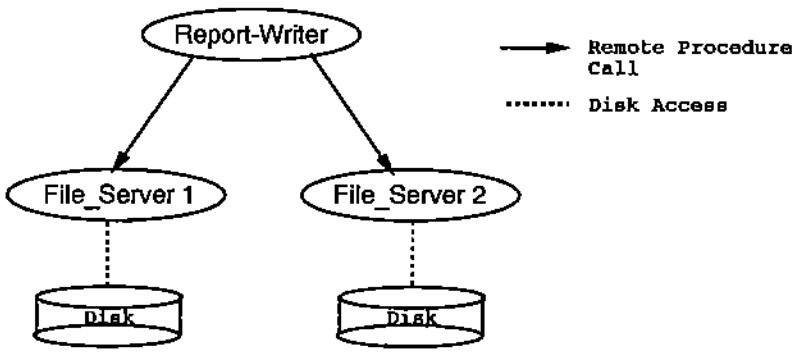

Figure 5: Process Layout for the Report-Writer application

processes as shown in Figure 5. The application accesses and updates the underlying data sources through the associated server process. As mentioned before Sun RPC mechanism is used for executing the methods of the server process.

The server processes for the data sources are created on demand. That is, when an application needs to access or update a data source, the corresponding server process is created. However, the process does not terminate on the completion of the method execution. Instead, once created the server process remains active as long as the application process is active. When the application process exits, the server processes associated with its data sources are terminated.

For data sources shared by multiple applications, one server per application is created. Thus each application has its own set of servers. The system classes do not provide any mechanism for controlling updates on a data source by multiple servers. However, when a fragment is accessed, the timestamp information is used to determine if the fragment contains most up to date data. If there are updates more recent than the timestamp, the fragment is re-read through the associated server process and the timestamp information is updated.

4.4 Discussion

The above Report-Writer application using the federated object approach. Currently, the application is written in C as the related X-window code and library uses C. The application uses C++ system classes for manipulation of data sources and runs on a network of Sun workstations under UNIX operating system.

The building of an application, even such a simple one, demonstrates the feasibility and usefulness of this approach. In future we will be rewriting the application in C++. We also plan to complete the implementation of *Fo_Server* system class and use that to build an application which requires sharing of both passive data (such as files) as well as data servers (database relations).

5 Conclusions and Future Work

We have presented a pragmatic approach for building applications that require access to multiple heterogeneous and distributed data sources. The two characteristic features of this approach are: i) data sources are not required to have full database system capabilities, and ii) the object-oriented approach is used for access and manipulation of the data sources. In this approach, building an application involves constructing a composite object (called federated object) from the desired set of data sources. Each data source is viewed as a fragment of the composite object. Methods can be defined to operate on both individual fragments as well as the entire composite object.

For each different type of possible data sources, a separate C++ class is defined, which

specifies the structure as well as the operations. A weaker mechanism for consistency between fragment and corresponding data sources is provided through the methods defined in the class corresponding to the data source. This weaker notion of consistency is desirable for applications such as those used for CSCW. Since the mechanism is built into the classes defined for data sources, they can always be replaced by adding code for stronger notion of consistency, if so desired.

Our initial experience of building a simple report writer application has demonstrated the feasibility of this approach. In future, we plan to extend the report writer application to involve access to both passive data (such as files) as well as access to data through a data server (e.g a database management system). An advantage of this approach is that it is evolutionary. After building a standard core library, enhancements to the mechanisms can be done quickly.

References

[ABGM90] R. Alonso, D. Barabara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[ASD+91] R. Ahmed, P. D. Smedt, W. Du, W. Kent, M. A. Ketabchi, W. A. Litwin, A. Rafii, and M. Shan. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, 24(12):19–27, December 1991.

[BBS92] B. Bhargava, S. Browne, and J. Srinvasan. Composite Object Replication in Distributed Database Systems. *Proc. International Conference on Information Systems and Management of Data*, July 1992. To appear.

[BDMS90] B. Bhargava, P. Dewan, J. G. Mullen, and J. Srinivasan. Implementing Object Support in the RAID Distributed Database System. In *Proceedings Of The First International Conference on Systems Integration*, pages 368–377, April 1990.

[BOH+91] A. Buchmann, M. T. Ozsu, M. Hornick, D. Georgakopoulos, and F. A. Manola. A Transaction Model for Active Distributed Object Systems. In G. Goos and J. Hartmanis, editors, *Lectures Notes in Computer Science:85.* Springer-Verlag, Berlin, 1991.

[DVB89] P. Dewan, A. Vikram, and B. Bhargava. Engineering the Object-Relation Model in O-Raid. In *Proceedings Of the International Conference on Foundations of Data Organization and Algorithms*, pages 389–403, June 1989.

[EN89] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin-Cummings, Menlo Park, Cali., 1989.

[FT88] G. Foster and D. Tatar. Experiments in Computer Support for Teamwork – Colab (Video), 1988. Xerox PARC.

[Gup89] A. Gupta, editor. *Integration of Information Systems:Bridging Heterogeneous Databases.* IEEE Press, 1989.

[KBG89] W. Kim, E. Bertino, and J. F. Garza. Composite Objects Revisited. In *Proceedings of ACM/SIGMOD International Conference on Management of Data*, pages 423–432, june 1989.

[MNS92] M. Makpangou, Y. Gourhant J. Le Narzul, and M. Shapiro. Fragmented Objects for Distributed Abstractions. *IEEE Transactions on Software Engineering*, 1992. To appear.

[MS90] A. Margaret and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Mass., 1990.

[MSDB90] J. G. Mullen, J. Srinivasan, P. Dewan, and B. Bhargava. Supporting Queries in the O-Raid Object-Oriented Database System. In *Proceedings of the International Computer Software and Applications Conference*, 1990. to appear.

[RS87] L. A. Rowe and M. Stonebraker. The POSTGRES Data Model. In *Proc. 13th VLDB Conference*, 1987.

[SG83] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 16(8):57–69, 1983.

[SL90] A. Sheth and J. L. Larson. Federated Databases: Architectures and Integration. *Computing surveys*, 22(3), September 1990.

[Sun88] Sun Microsystems. *Network Programming*, May 1988.

[TSB91] H. Tirri, J. Srinivasan, and B. Bhargava. Transactions for Fragmented Composite Objects. Technical Report CSD-TR-91-083, Purdue University, November 1991.