Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Computer Standards & Interfaces 25 (2003) 373–389

COMPUTER STANDARDS & INTERFACES

www.elsevier.com/locate/csi

Integration of CORBA and object relational databases

Kai-Chih Liang[a,*], Daphne Chyan[b], Yue-Shan Chang[c], Win-Tsung Lo[d], Shyan-Ming Yuan[a]

[a] *Department of Computer and Information Science, National Chiao Tung University, 1001 Ta-Hsieh Road, Hsinchu, Taiwan, ROC*
[b] *W&Jsoft Inc., 7/F 173 GuengYuan Road, Hsinchu, Taiwan, ROC*
[c] *Department of Electronic Engineering, Minghsing Institute of Technology, 1 Hsin-Hsing Road, Hsinfong, Hsinchu, Taiwan, ROC*
[d] *Department of Computer Science and Information Engineering, Tunghai University, 181 Taichung-kang Road, Sec. 3, Taichung, Taiwan, ROC*

Abstract

CORBA is widely accepted as the open international standard for modelling and building comprehensive distributed systems. In most cases, CORBA architects have adopted relational databases for storage of persistent data. Among the issues that usually face architecture designers considering how to combine CORBA and standard relational database standards are fault tolerance, performance, and the extensibility and scalability of the systems. The research team involved with this paper found that the ODMG object database concept is useful to solve the issues encountered when integrating CORBA and relational database standards. The reference architecture, which the team devises, integrates CORBA and relational databases without compromise on the necessary transactional properties. The CORBA standard object transaction service and concurrency control service are reused. The team also develop an object relational data modelling tool—Latte—that supports the overall design intention as well the development paradigms for the proposed architecture. The implementation of the system is useful to CORBA, ODMG, and relational database architects because it provides a unified modelling and programming paradigm capable of solving the problems of managing mission-critical distributed data. Thus, we present a case study of combining different international standards to build a comprehensive system.

Keywords: CORBA; ODMG; Object relational database; Object transaction service; Object concurrency control service; Fault tolerance system

1. Introduction

CORBA technology is now widely accepted as the standard for distributed system architecture. Along with the core architecture standard, CORBA specifications also include fundamental object services [10] as well as vertical and horizontal facilities [11]. This helps the architect to reuse the specific CORBA standards when modelling mission-critical systems. However, even though CORBA standards define the Persistent Object Service (POS) [12] as a reusable persistent management service, software architects can still not ignore the wide deployment base of relational databases and the ease of SQL programming technologies [3]. They must frequently adopt these databases into CORBA-based software architecture design. However, there is no standard or reference architecture for combining the two technologies. Soft-

* Corresponding author.
E-mail addresses: kcliang@cis.nctu.edu.tw (K.-C. Liang), daphne@wnjsoft.com (D. Chyan), ysc@mhit.edu.tw (Y.-S. Chang), winston@mail.thu.edu.tw (W. Lo), smyuan@cis.nctu.edu.tw (S.-M. Yuan).

0920-5489/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0920-5489(03)00009-6

ware architects facing this situation have to reconsider the feasibility and support of classical transactional properties (Atomicity Consistency Isolation Durability (ACID)) [4] as well as software availability and performance issues. Although CORBA standards also define the Objective Transaction Service (OTS) [13] and the Concurrency Control Service (CCS) [14] as fundamental supports for ACID properties, the process is still time-consuming and error-prone. Being able to reuse a proven combinational architecture model is of great benefit.

This paper proposes an integrated reference architecture as well corresponding implementation to address the issues about the combination of CORBA and relational database standards. Classical transactional properties, system fault tolerance and system load balance issues are taken into consideration in the architecture. In addition, the design features standard development paradigms. A software tool, Latte, is provided to support the object relational data modelling. The key design area of the proposed software architecture is the use of the ODMG object database concepts [15].

1.1. Hybrid object relational data management system development process

ODMG defines the standard reference model for the object data management applications. In the ODMG data model, data is represented logically by the ODMG data object. ODMG defines bindings between ODMG data model and popular object-oriented programming languages, such as C++ and Java. ODMG-compliant database users only need to manipulate the data via the programming language objects. This greatly reduces the gap between real data and programming language objects and is of benefit to the completion of OOA/OOD/OOP processes. Reusing concepts found in ODMG, this paper is intended to introduce the ODMG paradigms into the solution architecture of combining CORBA and relational databases. In this way, software architects are able to access the relational data either from CORBA or ODMG programming paradigms. They are also able to take this combination as a hybrid architecture pattern that provides new object data management practices similar to both CORBA and ODMG. Fig. 1 illustrates the hybrid development process for CORBA/ODMG/RDBMS integration.

The three phases in the hybrid development process are—Data Modelling, Logic Modelling, and Deployment. Latte is the object data modelling tool used to perform the first of these. To reuse the existing relational data model, Latte extracts the existing schema and defines the logical mapping between the relational data model and the object data model. Users can also define a new data model from the object data model, which Latte converts into a relational data

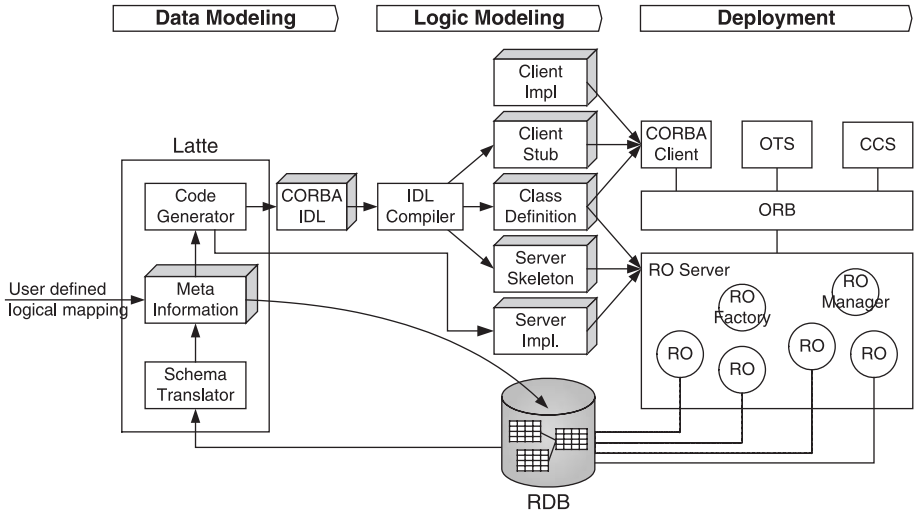

Fig. 1. Hybrid development process for CORBA/ODMG/RDBMS integration.

schema. When the data model is ready, Latte registers the schema in specific Meta data storage, which is typically a relational database. In the meantime, Latte also generates a data manipulation code and a corresponding CORBA IDL to represent the defined data model in the CORBA. The focus is then on data manipulation logic modelling by the use of new OOA/OOD knowledge. Note that Latte's code generator automatically creates data manipulation mechanisms such as load and store operations. Finally, the user compiles the program modules and deploys the components to the CORBA environment.

2. Previous works

Before work began on designing and developing the integrated object relational data management system based on CORBA described in this paper, the project research team had completed a series of steps to examine the necessary technologies and their feasibility. These are described below.

2.1. CORBA object services

The Object Transaction Service (OTS) and the Concurrency Control Service (CCS) are two CORBA common object services that guarantee transaction and concurrency semantics for CORBA objects. In a previous work, the research team had developed the OTS and the CCS [5,6] over Orbix ORB [2]. These services can be reused through their standard interfaces.

2.2. WOO-DB Java binding

An ODBMS makes database objects appear as programming language objects in one or more existing programming languages. ODMG defines the binding between the ODMG object model and native programming languages such as C++, Smalltalk, and Java. WOO-DB is an ODMG-compliant OODB developed by the Institute for the Information Industry and originally it provided C and C++ binding. The research team has sought to improve the connectivity of WOO-DB [1,8] by designing and implementing WOO-DB Java binding according to ODMG Java Binding Specifications [15].

2.3. Distributed object-based database architecture model

In most cases, the scope of transactions is limited to a single database. For transactions spanning multiple databases, other mechanisms such as a TP Monitor have been introduced. Following its experience with WOO-DB Java Binding, the research team proposes an architecture model [7] for a distributed object-based database that supports the ODMG [15] object model. This contains the key features of a distributed object management system that has transactions spanning multiple databases.

A further purpose of the model is to reduce the burden of maintaining the database architecture caused by the changes in the outside environment and application demands. The database architecture is based on components and relational database, which are built upon a distributed object infrastructure.

Fig. 2 reveals the distributed object-based database architecture model. Application objects are found, of course, in the application layer, at the top, and data objects in the middle layer. An application object accesses data from the data object through an interface opened by the data object. The data object is prepared, i.e., the data are defined or created by *data definition logic*. Data manipulation logic has the function of acquiring a data object, e.g., a query or a transaction. Since placing related objects together achieves better performance, we have included a *service/data object container*. To support the dynamic requirements of data semantics, the research team separates the semantics from data objects. Below the logic layer comes the semantic layer. Data objects delegate their semantics requirement, such as relationship, concurrency, and persistent store, to semantic objects in the semantic layer. There are certain functions that do not belong to any of the above three layers, such as naming, transaction, and life cycle and these are grouped in the Scope Management System that spans the three layers vertically.

2.4. CORBA/OODB integration

To prove the feasibility of our distributed object-based database architecture model, the research team had done a work to integrate CORBA and ODMG-compliant OODB [16]. OODB developers define

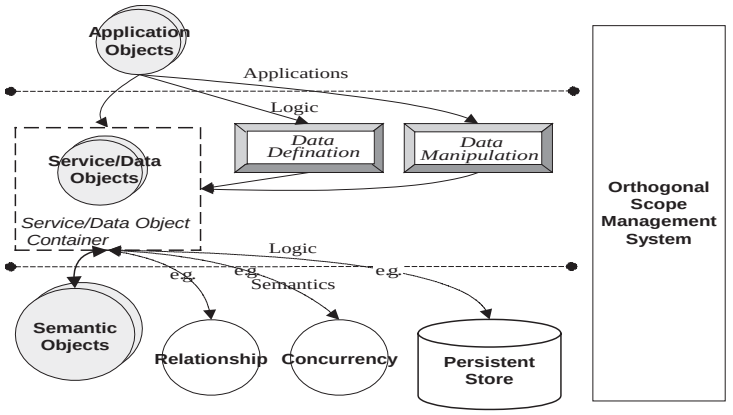

Fig. 2. Distributed object-based database architecture model.

objects by ODL and are able to implement them with no need of any CORBA domain knowledge, as if they were implementing pure OODB objects. CORBA client programmers, for whom object definitions are translated into CORBA IDL, are able to develop clients in their normal way. To reduce the burden of incorporating CORBA with OODB, the associate object implementation code is automatically generated. This provides access to data in database and the transaction and concurrency control functions.

3. System model design

The system serves as a logical object-oriented relational database (ORDB) in a distributed object environment. How to map the relational data model into the object data model is a key question for ORDB system designers. A further question is how to integrate standalone legacy RDBMs, so we decided to introduce into our design such features as transactions spanning multiple databases and concurrency control. The integration can be illustrated in different models.

3.1. Three-tier model

It is the limitations of two-tier architecture (client/server architecture) that spurred the development of three-tier architecture [9], which components are separated into three layers, namely, the presentation layer, the functionality/business layer, and the data layer. The benefits of the three-tier architecture are the ability to partition application logic, capabilities of system robustness, scalability, as well as single user image. These meet our requirements of our system design. Fig. 3 shows how we separate the components of our system architecture into three layers. CORBA application objects are mapped into the presentation layer. In the data layer, all the states of objects are stored in the relational databases. The data manipulation logic, that is, the logic to retrieve and to access data, transaction, and concurrency control, resides in the functionality/business logic layer.

3.2. Data model

The relational data model and the object-oriented data model differ in fundamental modelling philosophy. To integrate the first with object-oriented logic requires a proper mapping of relations into object-oriented technology. We begin by simply mapping

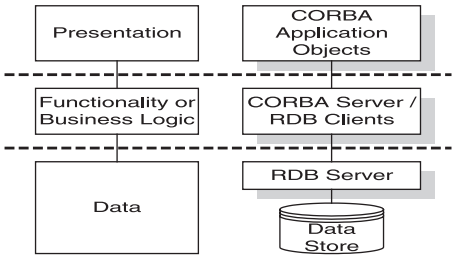

Fig. 3. The three-tier model.

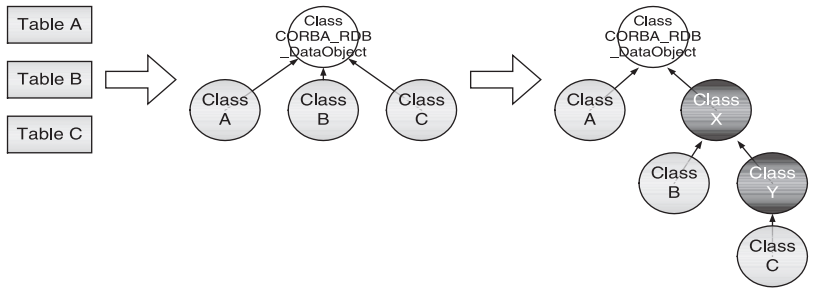

Fig. 4. An example of user-defined inheritance hierarchy.

each table in the relational model into a primitive class. Classes of this type all inherit from a common base class, *CORBA_RDB_DataObject*. The desired inheritance hierarchy among the classes is constructed with any one of three operations: *single_inherits_from_new*, which allows an existing class to inherit from a new class; *single_inherits_from_existinng*, which allows an existing class to inherit from another existing class; and *multiple_inherits_from_new*, which allows specified multiple classes to inherit from a new class.

Fig. 4 illustrates an example of constructing object types from relational tables. Classes *A*, *B*, and *C* are primitive classes derived from relational tables, while classes *X* and *Y* are new classes added by user.

Data type mapping between the database and the CORBA IDL is also necessary. In addition, to reduce the complexity of the system, each row of data is associated with one data object only. The decision on this design strategy is fundamental to maintaining data consistency.

3.3. Transaction model

Since our system serves as a multidatabase system in a distributed object environment, a mechanism to manage transactions among databases is needed. CORBA OTS is a transaction manager that coordinates transactions across multiple processes, threads, or spans more than one logical database.

Fig. 5 illustrates the transaction model in our RDBs/CORBA integration architecture. There are multiple *recoverable objects* (ROs) within a CORBA transaction that are managed by a *Coordinator object*. The data associated with the ROs span more than one database and the *RO Servers* are distributed. Within an RO Server, the ROs participate in the same OTS transaction and associate with the same database to

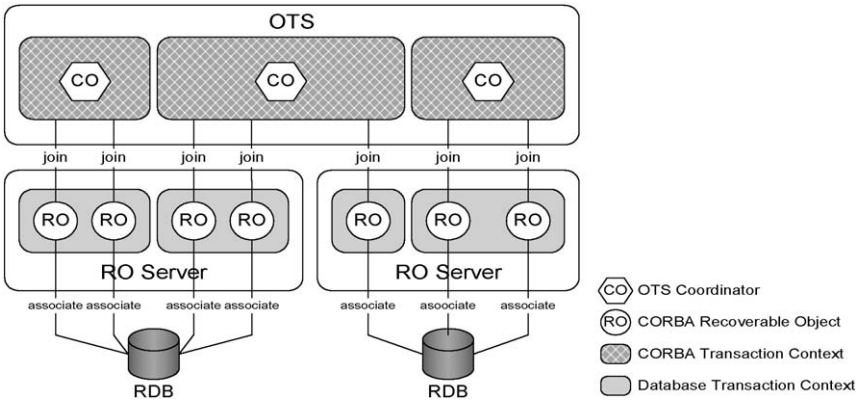

Fig. 5. The transaction model for RDBs/CORBA integration.

form a group. The RO Server has the task of creating a local database transaction for each group within it. These local database transactions may be viewed as the nested transactions of the CORBA transaction.

Data objects in the system are all shared resources. There may be several users accessing the same objects at the same time. So we use CCS to guarantee consistency for concurrent access. It is obvious that the granularity for concurrency control is an object. The system uses three of the lock modes defined in CCS: *read*, *write*, and *upgrade*. To avoid deadlock, our system provides some constraints. To obtain data for later updating, the method is *get_upgrade_*<*attribute name*>and for read-only data, the method is *get_readonly_*<*attribute name*>.

3.4. Failure model

Although failure seldom occurs, the system still has to keep objects and databases in a consistent state if it happens. There are two possible kinds of system failure here:

1. RO Failures: The effect of an RO failure is limited to the object itself. It is detected on request from clients. The states of the object are automatically recovered from the permanent storage.
2. RO Server Failures: When an RO Server fails, it influences all the CORBA clients and the states of the ROs within the server in question. We must keep them all in a consistent state. So when a failure is discovered, we have to REDO or UNDO all operations on the objects.

The recovery process for an RO Server is initiated when the RO Server is restarted from failure. Another case is for the client to request the ROs that originally resided in a failed RO Server. The recovery process is also initiated.

The point in time at which RO Server fails, it decides whether a transaction should be REDO or UNDO. Our failure mode in Fig. 6 shows that the *OTS recovery point* is the boundary between REDO and UNDO. As the OTS receives all of the return votes from participating resources, it makes the final decision to commit or roll back the transaction. The point in time when the OTS makes the final decision is the recovery point.

For a transaction, if the RO Server fails before the recovery point, we have to UNDO all operations, whereas if the failure occurs after the recovery point, we have to REDO or UNDO the operations depending on what the final decision is. As long as the final decision is *Vote*::*commit*, all operations must be made persistent. In contrast, if the final decision is *Vote*::*rollback*, all states must be rolled back. We detail our implementation issues later.

4. System architecture design

4.1. System overview

Fig. 7 is the functional diagram for the components of RDBs/CORBA integration architecture. They are CORBA client, RO Server, CCS, OTS, and RDBs that support the ODBC interfaces. As can be seen in the three-tier model introduced above, these components are separated into three layers. The interaction between the CORBA client, the OTS, the CCS, and the RO Server is through the CORBA Interface and the RO Server accesses the RDB through the ODBC interface.

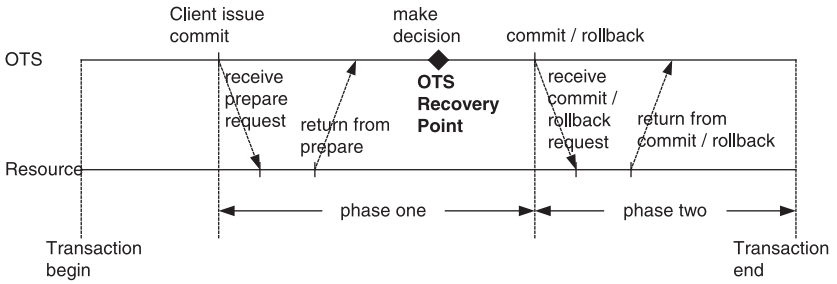

Fig. 6. The diagram of RO Server failure.

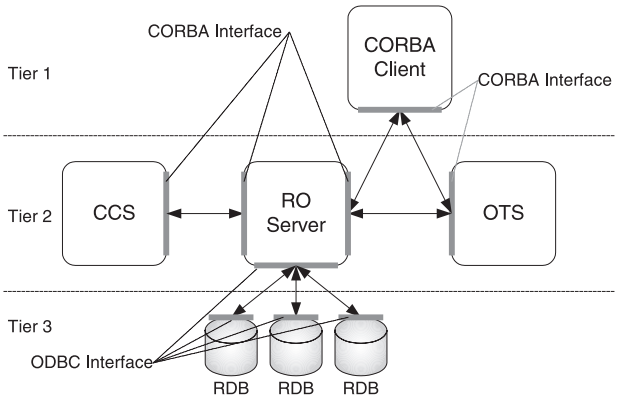

Fig. 7. System components of RDB/CORBA integration architecture.

Within the RO Server, each RO is associated with a row of data in an RDB. We wrap the data manipulation logic and the recovery function into the ROs. Apart from the RO objects, there are the RO Factories and the RO Managers in the RO Server. They provide the logic function for data object retrieval and transaction management within the RO Server, while the ROs are persistent data objects. The OTS is a transaction manager whose task is to passively begin/end a transaction. The concurrency control for the shared ROs is provided by the CCS. The CORBA client is an object that manipulates persistent data objects within transactions. The RDBs supporting the ODBC interface are pluggable and serve as persistent storage for data objects.

4.2. System architecture

4.2.1. Latte

The word ‘‘Latte’’ refers to a kind of hybrid coffee. Since the tool in our design combines several functions, we take our cue from ‘‘hybrid’’ and so name our tool Latte.

In our RDB/CORBA integration architecture, Latte is a realization of the code generation model we described above and provides a GUI tool (see Fig. 8). It hides all the tasks for mapping the relational data model into the object data model. Those tasks are:

1. Schema translation: Schemas are the logical data structures. Different data models may have differ-

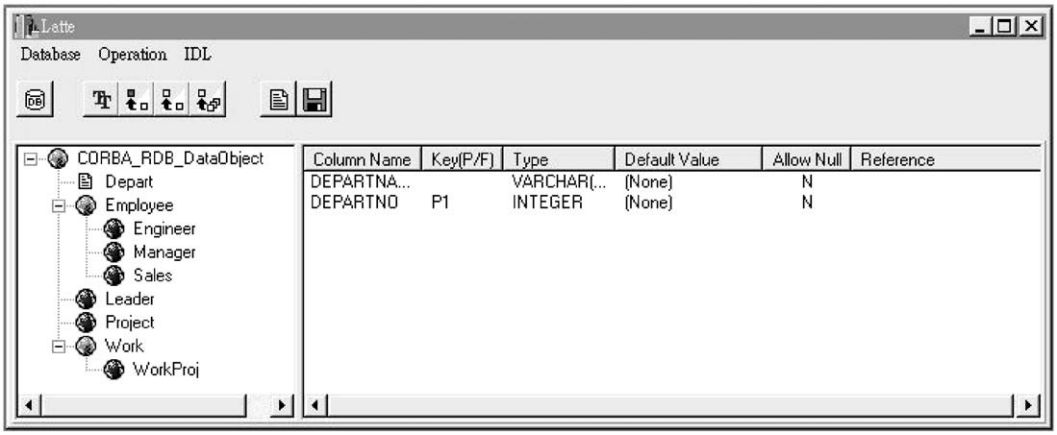

Fig. 8. Latte object relational data modeling tool.

ent schemas. Latte carries out the task of translating relational schema into object schema.
2. User-defined inheritance hierarchy: We provide a GUI tool to allow users to define their own mapping between relational data and object data. It has three operations: single_inherits_from_new; single_inherits_from_existing, and multiple_inherits_from_new.
3. Meta data storing: The user-defined mapping information between relational data and object data is called *Meta* data. This should be stored in permanent storage such as databases, so that information can be retrieves on demand.
4. CORBA IDL: Latte defines interfaces for objects. These interfaces are described by CORBA IDL, which are stored into files.
5. RO component implementation: Latte hides the task of integrating the RDB and the CORBA. We wrap the function of accessing relational data into *RO objects*. The transaction semantics and concurrent access functions are also wrapped into RO objects. The necessary codes for RO objects and those for the *RO Factory* and the *RO Manager* are automatically generated by Latte, as is the RO Server program.

4.2.2. RO components

The RO components comprise the RO Servers, the RO Factories, the RO Managers, and the ROs. Since they are closely related, we put them together. Each of these components is detailed as follows.

4.2.2.1. RO Server. We conceptually partition the CORBA Server address space into several RO Servers (see Fig. 9). Each one is associated with an *RO Factory object*. All the RO objects and the RO Manager objects created by a given RO Factory object also reside in the RO Server corresponding to the given RO Factory. If an RO Server failure occurs, all the object states in that server must be recovered when it is restarted or when any other RO Server detects the failure. In addition, to retain performance, each RO Server has a memory cache that records all the RO objects within the server and associated transaction information.

4.2.2.2. RO. An RO object is a data object. 'R' stands for "Recoverable", indicating that this kind of object has the capability to recover. When a failure occurs, an RO object can regain its state, i.e., remain consistent. When a transaction is completed, the state of the object including its data must remain in a consistent state.

4.2.2.3. RO Factory. In the system, the RO Factory is responsible for creating RO objects. Each type of RO is associated with a type of RO Factory. That is, RO Factories of different types are responsible for creating ROs of different types. Thus, all ROs in the same RO Server are associated with the same data type and their persistent data are stored in the same database. An RO Factory retrieves data from a database according to the criteria specified by the user and wraps each data as an RO object. Apart from that, it provides the function of creating a new persistent object within a transaction.

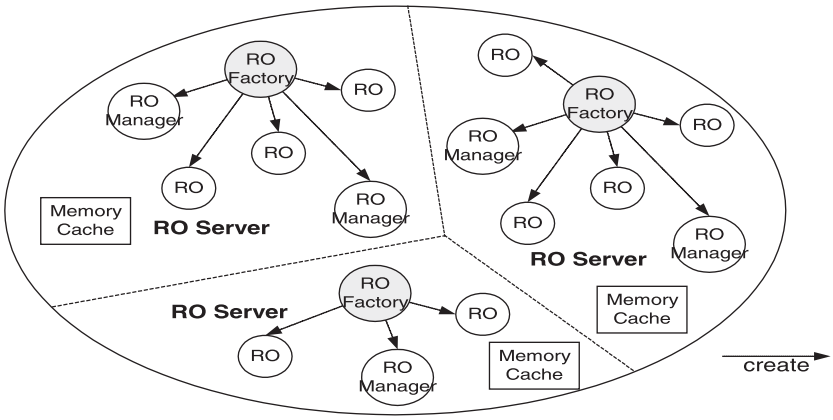

Fig. 9. The united RO Servers.

The data of this persistent object are stored into a database when the transaction is committed.

To reduce the complexity of maintaining data consistency among the RO objects associated with the same row of data in a database, each row is associated with just one RO object. So we must keep RO object references and some necessary information in persistent storage, such as in a database. Therefore, clients requesting ROs according to the same criteria obtain the same set of ROs. When a CORBA client requests an RO object, the RO Factory checks whether there is an RO associated with this data. If so, the RO Factory returns the object reference to the client. If not, the data is wrapped into a new RO object and retained in a log.

The RO Factory also has the task of dispatching transactions. It is responsible for mapping a CORBA transaction context into database transaction context within the spanned databases. We detail this task in the next section.

4.2.2.4. RO Manager. Although we claim that an RO object has the capability to recover, it does not implement the *CosTransactions*::*Resource* interface. It does not directly register itself as a resource in the OTS. Instead, an RO Manager serves as a mediator between the OTS and the RO object (see Fig. 10). In our design, each CORBA transaction is associated with an RO Manager object for each RO Server it spans. The manager is responsible for controlling the processing of certain transactions within the RO Server. The manager implements the CosTransactions::Resource interface and registers itself as a resource in the OTS. When the OTS tells the RO Manager to prepare/commit/rollback the transaction that it is associated with, the manager informs all the ROs in the same RO Server and participates in certain transactions to process the prepare/commit/rollback task.

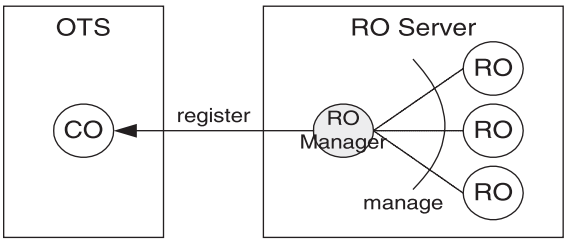

Fig. 10. The RO Manager.

4.3. Interactions of transactional programs

The interactions of transactional programs are demonstrated in Fig. 11. The client first binds to the desired servers, e.g., the OTS and the RO Factory, to serve the further requests. A new transaction is then initiated and a lookup invoked for the bound RO Factory with the specified criteria. The RO Factory creates a new RO Manager to regulate the transaction processing in the given RO Server. The manager registers itself as a resource in the OTS. A database transaction is also requested. Each row of the query results participates in the database transaction. The RO Factory checks whether the query results have been wrapped as ROs. Here, *a* has been wrapped, while *b* has not been wrapped yet. Therefore, the RO Factory checks out RO *a* and creates RO *b* to wrap the data in *b*. RO *a* and RO *b* both register themselves in the RO Manager. One manipulates the data of the ROs through transactional operations and then instructs the OTS to commit the transaction and the *two-phase commit protocol* starts.

In the *prepare* phase, OTS notifies the RO Manager, which then informs all the registered ROs to do the preparatory task, i.e., write their operation results into the database and make logs. It then returns the vote to the OTS. In the commit/rollback phase, the OTS notifies the RO Manager to commit/rollback. The manager also informs all registered ROs to perform the commit/rollback task, perform database commit/roll back operation, and clear the log.

5. Issues and constraints

5.1. Critical issues

5.1.1. RO Server recovery

To ensure the ACID properties, there must be some mechanism to deal with transient system failures. For recovery reasons, we must retain the state of both the CORBA transactions and the ROs in permanent storage. In this way, when the RO Server is restarted after a failure, it can recover the state of the CORBA transactions.

When a client invokes the RO's set_<attribute name>method, we update only the data in the RO's memory. In the prepare phase, these modified data are

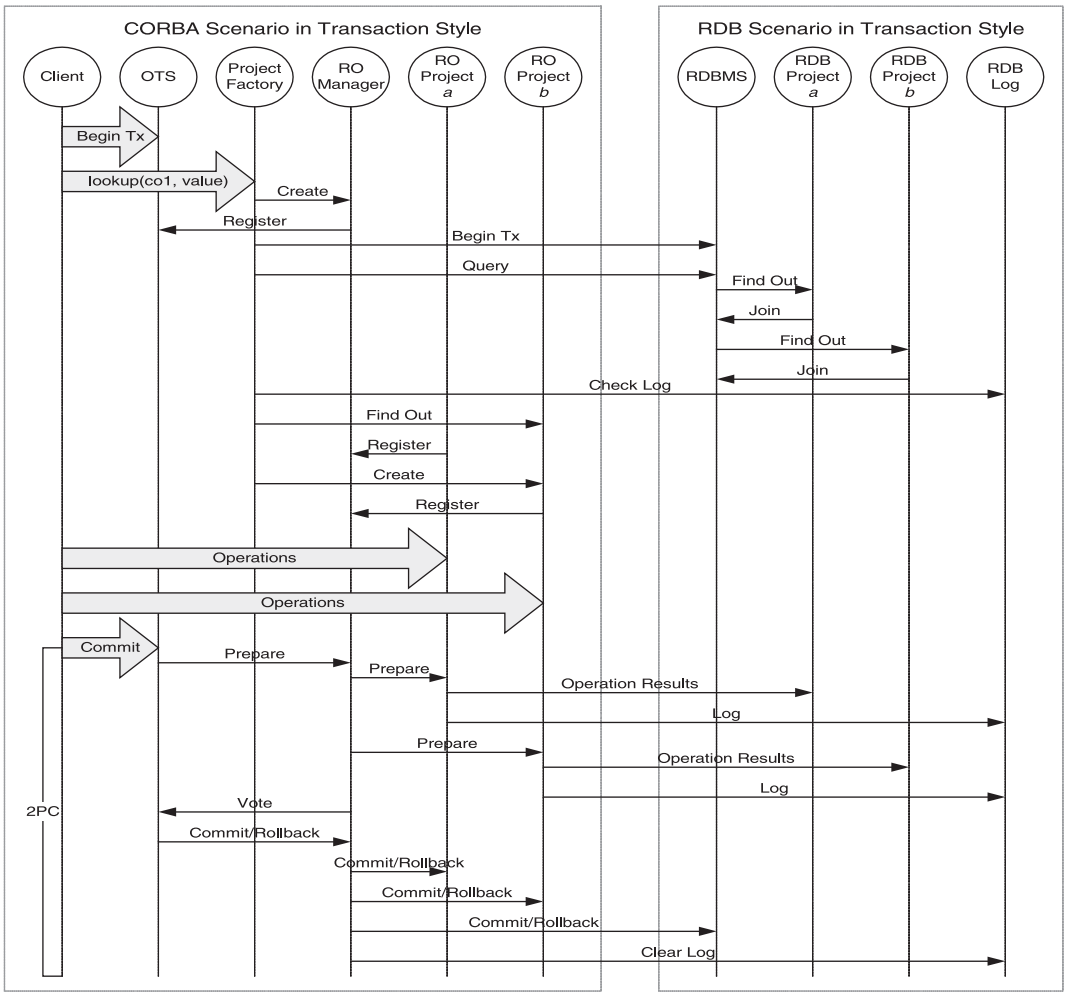

Fig. 11. The detail interactions in a transactional program.

written into the database within a database transaction. Thus, when the user manipulates RO data, we must retain the information about the type of operation (create new, delete, or update) for the RO.

As we described in Section 3, when an RO Server fails, if a CORBA transaction occurs before the OTS recovery, it must abort, whereas if it occurs after the point, it must commit or rollback, all accordance with the OTS's final decision.

Let us consider the effects of an RO Server failure. If this happens, all the database transactions on the server are automatically rolled back. A problem occurs where the OTS's final decision is *Vote*::*commit* but where the RO Server has failed before the corresponding database transaction commits. So, in the preparatory phase, apart from writing RO data into the database, we must also copy the data into persistent storage, such as in a database or in a log. Note that this log should be made persistent. When an RO Server is restarted after failure, it can obtain the RO's state from the log and recover the data involved in the transaction to the database. But how do know what state a transaction is in? The RO Manager must place checkpoints in the persistent storage with which the manager can restore the transaction state when the RO Server is restarted. Such checkpoints are shown in Fig. 12.

At the beginning of a transaction, the RO Manager places *checkpoint 1* and records the object reference

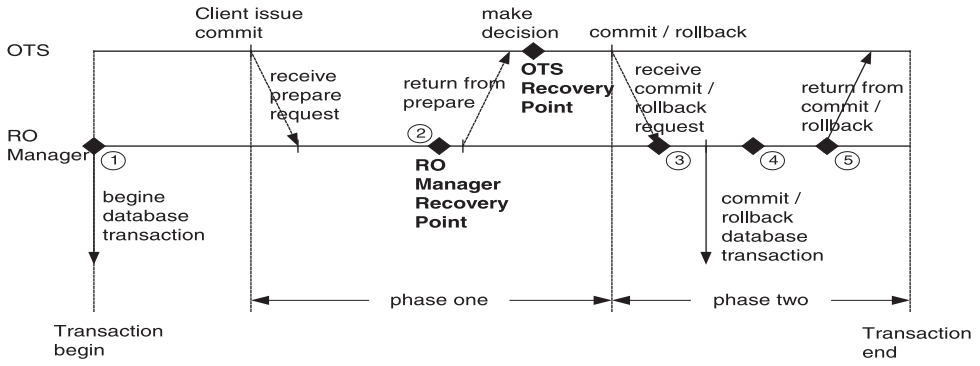

Fig. 12. The checkpoints for a transaction.

strings of the OTS *Coordinator* and *RecoveryCoordinator* corresponding to the transaction. At the same time, the database transaction also begins. When the manager collects all the RO votes participating in the transaction, it makes a local decision based on the votes and returns it to the OTS. This local decision must be retained in *checkpoint 2*. When the manager receives the *commit* or *abort* request from the OTS, it places *checkpoint 3* to confirm recognition of OTS's final decision. The manager then commits or rolls back the corresponding database transaction and places *checkpoint 4* to specify that the database transaction has finished. At *checkpoint 5*, all the logs associated with this transaction in this RO Server are removed.

For a transaction, if the RO Server fails between checkpoints 1 and 2, then clearly the transaction is rolled back. In the case of failure between checkpoints 2 and 3, if the local decision is "prepare fail", then the transaction is rolled back. But if the decision is "prepare succeed", since we do not know the final OTS decision, we must ask the OTS for it by issuing a *replay* request to the corresponding OTS RecoveryCoordinator, which can be retrieved from the log. In the case of failure between checkpoints 3 and 4, and if the decision is *Vote*::*commit*, we can restore the ROs' prefailure states from the backup log. These states must be restored to the database. Finally, if the failure occurs between checkpoints 4 and 5, the only thing we must do is to clear the log associated with the transaction.

5.1.2. Keys

Primary keys and foreign keys are important issues for relations in the relational model. Their function, working in pairs, is to automatically construct relations between data. The function of unique keys is to specify the uniqueness of attribute values. Since there is no clear concept about keys in an object model in our system, we simply treat all key attributes as normal. The mapping of both key attributes and non-key attributes into object attributes is performed in the same manner.

On the other hand, since the back-end storage is the relational database, certain constraints on key attributes remain, just as there remains for those in the relational model. The constraints are the uniqueness of the primary and unique key attributes and the limitations on updating the values of all the key attributes, etc. To reduce the complexity of the system, we delegate checking these integrity constraints to the RDBMSs and collect the exceptions from back-end databases.

5.1.3. Data modification propagation

When Latte is used to build an inheritance hierarchy, a new class that we term a *nonoriginal table* class because it does not correspond to any existing table in a database, can be created. Should we provide nonoriginal table class persistent storage? If we did so, the modification of data in one class would perhaps propagate to other classes. This is an issue worthy of discussion.

If we provide persistent storage for all classes, any modification of data in one class could perhaps propagate to its ancestors and even to its descendants if it has any. But in some cases, such modification could propagate not only to ancestors and descendants, but also to siblings. This would usually be due to the overlapping of data among sibling classes. Since we do not know the purpose of the original table

schema, we cannot automatically tell whether in fact the modification will propagate to siblings. Therefore, at this stage we do not provide persistent storage for nonoriginal table class, which makes them virtual classes. At most, any modification will propagate to ancestors and descendants only.

5.1.4. Multi-inheritance

Since the CORBA IDL supports multi-inheritance, an interface can inherit from more than one interface. But if we support multi-inheritance, the problem of modification propagation becomes more complex. Therefore, at this stage, we only support single inheritance.

5.2. System constraints

5.2.1. Transactions

To guarantee correctness, we stipulate that all RO objects may be used only within the transaction in which they were requested. This concept is similar to the one for the database transaction.

Although the user may manipulate an RO object several times within a transaction, the final value is written into the database only once the transaction ends. A problem can arise if the operation order gets lost within the transaction. A possible solution to this is to record every operation performed within the transaction in question, but that solution is too expensive. So, to achieve higher performance, we just write the final value into database when the transaction ends.

5.2.2. Working together with legacy applications

In normal situations, applications in our system can work alongside legacy applications. Concurrent access is guaranteed by the database transactions. But if an RO Server fails, the database transactions are automatically rolled back and the database locks for the data in question are released, with a resulting, perhaps insoluble, problem.

A possible problem can arise where the final decision of an OTS is *Vote*::*commit* but where the RO Server fails before the corresponding database transaction commits. In our system, any failed transaction is automatically recovered later, and any application that accesses nonconsistent data is blocked until the recovery process is completed. The ACID properties are preserved. But if there is access through legacy applications to the nonconsistent data directly before the recovery process, there is no guarantee about data consistency because database locks have been released.

In Fig. 13, the CORBA transaction spans two RO Servers A and B. In the commit phase, Server A fails before the database transaction is completed and Server B succeeds in committing the database transaction. At this time, the data in database a is in pretransaction state, and that in databases a and b may be inconsistent. If legacy application X accesses the data in database a before the beginning of the recovery process for Server A, it retrieves the older version of the data, in which case a version control problem occurs. If legacy application Y accesses the

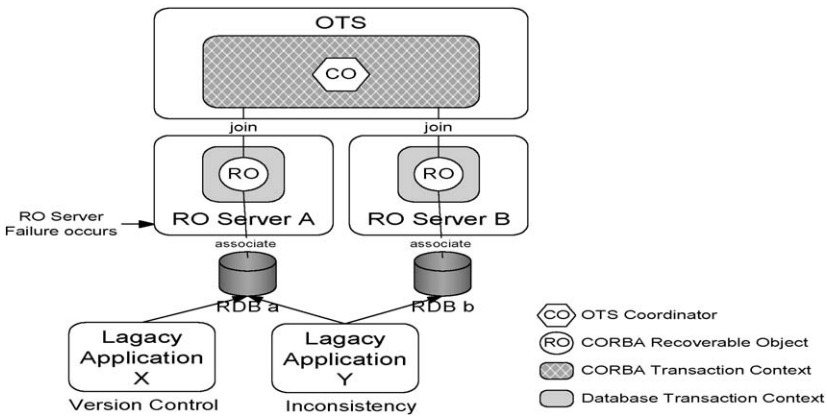

Fig. 13. The problems of version control and inconsistency.

data in databases a and b before the recovery process for Server A starts, it retrieves the inconsistent data, in which case an inconsistency problem occurs.

5.3. System performance

In measuring performance, since what we are mainly concerned about is the overhead in our system, the data schema and population in back-end databases are ignored. Only certain critical operations to prove whether the performance of our integration system is reasonable are tested.

To evaluate the system performance, we deploy three servers, OTS Server, CCS Server, and RO Server. The OTS Server and the CCS Server provide the transaction semantics and concurrency control functionality. The RO Server is responsible to access database, warp ROs, and vote back to OTS in the two-phase commit protocol (2PC).

The testing involves deploying three servers, the OTS Server, the CCS Server, and the RO Server. The functions provided by the OTS and CCS Servers are transaction semantics and concurrency control. The RO Server is responsible for accessing the database, wrapping ROs, and sending votes back to the OTS in the two-phase commit protocol (2PC).

Table 1 illustrates the testing environment and Table 2 the testing database schema.

The system performance is measured by the response time that clients observe. In addition, we test how retrieval and update operations perform. As a 'warm-up' to the measurement, a number of light weight test cases are tried beforehand to ensure that our system has reached a stable state.

5.3.1. Retrieval

In our system, clients issue a lookup request with user-specified criteria to retrieve data objects. When this is done, the data ensures that the specified criteria are retrieved from the database and wrapped as ROs. In addition, to ensure data consistency, we log the transaction information, with a log for each RO. The performance evaluation for retrieval is shown in Fig. 14a and b.

Table 1
The testing environment

Programs	OTS, CCS, RO Server, Client
RDBMS	IBM DB2 5.0
CPU	AMD K6 II (416MHz)
Memory	128 MB
Operating System	Windows NT Server 4.0 Service Pack 5

Table 2
The testing database schema

```
Table Performance
{
  P int (Primary key),
  Q int
}
```

Here, we define two terms, *cold cache* and *warm cache*. The first means Cache Miss, which means that none of the data retrieved has been wrapped as an RO in any memory cache and that, therefore, a wrapping process for data retrieval is needed. The second means Cache Hit, which means that each required data has been wrapped as an RO in a given memory cache. With warm cache, we can simply take the ROs and return them. In contrast, cold cache takes a lot of extra time to wrap the ROs. In general, more time is spent on retrieving data objects with cold cache than with warm cache.

Because we have to request a *LockSet* for each RO, cold cache spends approximately a quarter of the time on communicating with the CCS. To reduce the overhead, we substitute an in-process CCS for the out-process CCS. The substitute can be reached by a dynamic link library or shared library, thereby reducing the total cost for each lookup operation by almost a quarter.

Although in the distributed object environment, the locations of objects are transparent, placing related objects together can improve performance. Therefore, we measure how warm cache performs in retrieving local and remote ROs. When a client issues a lookup invocation to an RO Server, any retrieved ROs residing in the server in question are local ones and any residing in other servers are remote ROs. We must make a validity check for each remote RO to ensure that no RO Server failure influences the consistency of the data retrieval. In addition, a one-way invocation to the remote server is also needed to check whether a particular RO should participate in a particular

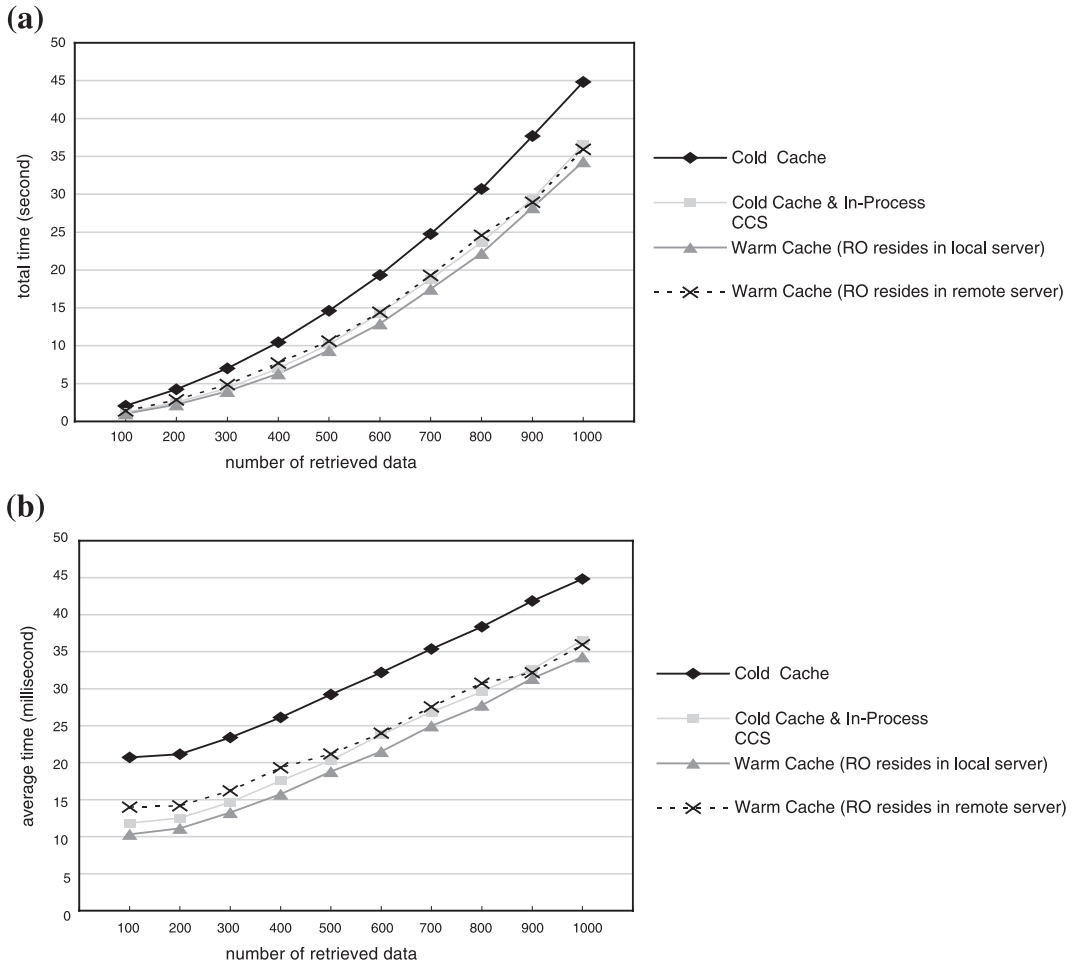

Fig. 14. (a) Total time of retrieve. (b) Average time of retrieve.

CORBA transaction. Since validity checking and one-way invocations lead to overhead, the performance for local ROs is better than that for remote ROs.

Fig. 14a and b shows that the total time curve is exponential and average time curve is linear. This is because we have to check and make a log for data retrieval. As the data retrieval set grows, so too does the time spent in checking and logging.

5.3.2. Update

In the architecture design, we delay the database update to the two-phase commit period. The client issues *CosTransactions*::*Terminator*::*commit()* and the CORBA transaction enters the commit period, the duration of which is the response time of *CosTransactions*::*Terminator*::*commit()*. We compare the update performance for our system and ODBC and show the results in Fig. 15a and b. It can be seen that as the influenced data set grows, the total update time for our system and ODBC both grow linearly, and so too does total overhead.

Since the commit request to the database and the communications between the client, the OTS, and the RO Server do not grow along with the data set, the overhead can be shared by the influenced data set. Consequently, the curves for the average time taken by our system and the ODBC converge to 6 and 1.9 ms, respectively. The average overhead of update also con-

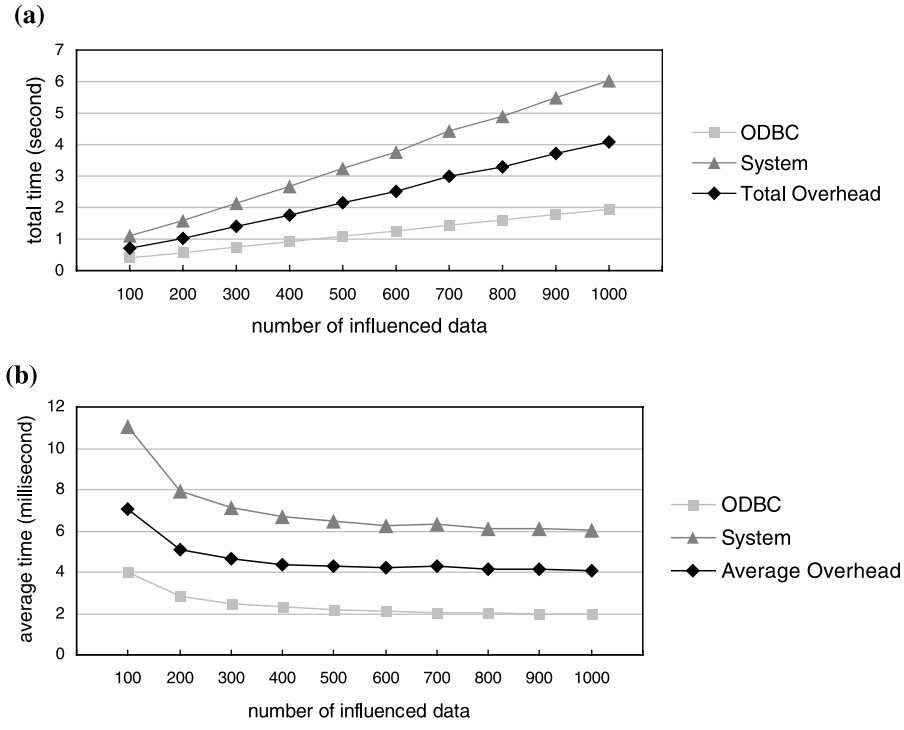

Fig. 15. (a) Total time of two-phase commit. (b) Average time for two-phase commit.

verges to 4 ms. Therefore, we can say that the update performance evaluation for our system is reasonable.

5.3.3. Discussion on improvements

Since it appears that the performance for date retrieval in the system is not perfect, we offer some suggestions for future improvements. We know that checking validity and logging cause retrieval bottlenecks, so we suggest delegating the logging task of the RO Server to another Log Server by issuing an asynchronous call to perform log operations for all data objects retrieved within a single lookup. Since this call is asynchronous, the RO Server can continue the processing after issuing the call. Note that we must aim to guarantee that the behavior of the new approach is the same as the original one.

6. Concluding remark—aggregating standards

OMG/CORBA and ODMG are two organizations committed to distributed object and object data management international standards. They both reinforce the completeness of the architecture models as well as the standards for reusability. However, relational databases have dominated the data processing market for more than two decades. Large amount of deployment base makes relational database irreplaceable. Software architects who take advantage of well-defined, distributed computing architecture and adopt existing data assets face their greatest challenge to aggregating standards. Because of the aggregation, several key design issues have to be reexamined.

This paper has proposed novel reference architecture to address those issues. Our design preserves the key ACID properties by adopting the CORBA OTS and CCS standards. By synchronizing the interface protocol between the system containers, it achieves high availability and scalability. A reference development process is available for streamlining activities when applying the reference architecture. The aggregated potential of these novel standards, CORBA, ODMG, and RDBMS, lies in the presence of the

framework for distributed object relational data management. This, in the future, can achieve object relational data management standards.

References

[1] H.C. Liao, Java Binding Based on WOO-DB, Department of Computer and Information Science, National Chiao-Tung University, Hsin-Chu, Taiwan, 1998.

[2] Iona, Orbix (http://www.iona.com/products/orbix3_home.htm).

[3] Information Technology—Database Languages—SQL (ISO/IEC 9075, 1992) (can also be found at http://www.jcc.com/SQLPages/jccs_sql.htm#SQL%20Publications).

[4] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Mateo, 1992.

[5] K.C. Liang, Transaction and Concurrency Control Services on CORBA, Master thesis, Department of Computer and Information Science, National Chiao-Tung University, 1996.

[6] K.C. Liang, S.M. Yuan, D. Liang, W. Lo, Nested transaction and concurrency control services on CORBA, Proceedings of Joint International Conference on Open Distributed Processing and Distributed Platforms 1F1PTC6, 1997, pp. 236–247, Toronto.

[7] K.C. Liang, S.M. Yuan, A Distributed Object Database Architecture on CORBA, Technical Report, Department of Computer and Information Science, National Chiao-Tung University, Hsin-Chu, Taiwan, 1998.

[8] K.C. Liang, S.M. Yuan, H.C. Liao, R.K. Sheu, W.J. Lee, J.C. Dai, C.H. Chen, C.H. Cheng, When Java Applet Meets Object Database, 7th WWW Conference, 1998.

[9] N. Jenkins, et al., Client/Server Unleashed, Sams Publishing, Indianapolis, IN, USA, 1996.

[10] Object Management Group, CORBAservices Specification (http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm).

[11] Object Management Group, CORBAfacilities Specification (http://www.omg.org/technology/documents/corbafacilities_spec_catalog.htm).

[12] Object Management Group, Persistent Object Service (http://www.omg.org/cgi-bin/doc?orbos/99-07-07.pdf).

[13] Object Management Group, Object Transaction Service (http://www.omg.org/cgi-bin/doc?formal/00-06-28.pdf).

[14] Object Management Group, Concurrency Control Service (http://www.omg.org/cgi-bin/doc?formal/00-06-14.pdf).

[15] R.G.G. Cattell (Ed.), The Object Database Standard: ODMG 3.0, Morgan Kaufmann Publishers, San Francisco, 2000.

[16] R.K. Sheu, K.C. Liang, S.M. Yuan, W.T. Lo, A new architecture for integration of CORBA and OODB, IEEE Transactions on Knowledge and Data Engineering 11 (5) (1999) 748–768.

Kai-Chih Liang received his BS and MS degrees in computer and information science from National Chiao Tung University, Taiwan in 1994 and 1996, respectively. He is now the PhD candidate in computer science of the same school. His current research interests include Web technology, distributed object computing architecture, high-confidence middleware, enterprise application integration, and software engineering.

Chii-Hwa Chyan received her BS and MS degrees in computer and information science from National Chiao Tung University, Taiwan in 1997 and 1999, respectively. She is an active consultant in the W&Jsoft, Taiwan. Her primary industrial domain is manufacturing and e-Business. Her current research interests include Web technology, business process automation and integration, workflow technology, and software engineering.

Chang Yue-Shan was born on August 4, 1965 in Tainan, Taiwan, Republic of China. He received his BS degree in Electronic Technology from National Taiwan Institute of Technology in 1990 and his MS degree in Electrical Engineering from the National Cheng Kung University in 1992. Currently, he is a graduate student of PhD degree in Computer and Information Science at National Chiao Tung University. His research interests are in distributed systems, object-oriented programming, fault tolerance, and computer network.

Win-Tsung Lo received his BS MS degrees in applied mathematics from National Tsing Hua University, Taiwan, Republic of China, and his PhD degree in computer science from the University Of Maryland. He is now an associate professor of Computer Science and the director of Computer Center at Tung Hai University, Taiwan, Republic of China. His research interests include architecture of distributed systems, data exchange in heterogeneous environments, and multicasts routing in computer networks.

Shyan-Ming Yuan was born on July 11,1959 in Mauli, Taiwan, Republic of China. He received his BSEE degree from National Taiwan University in 1981, his MS degree in Computer Science from University of Maryland Baltimore County in 1985, and his PhD degree in Computer Science from University of Maryland College Park in 1989. Dr. Yuan joined the Electronics Research and Service Organization, Industrial Technology Research Institute as a Research Member in October 1989. Since September 1990, he had been an Associated Professor at the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan. He became a Professor in June 1995. His current research interests include distributed objects, Internet technologies, and software system integration. Dr. Yuan is a member of ACM and IEEE.