Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Using Traders for Loosely Integrating Heterogeneous Database Systems

Wanlei Zhou, Philip Hepner and Xidong Wang

School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

Abstract

This paper presents the design and prototype implementation of a loosely integrated heterogeneous database system. The main goal of the design is to let local databases maintain full autonomy over their databases, yet when willing, they can share some portion of their information. These local databases register with a trader the portion of information that can be shared by using service offers. A service offer can be in the form of data, operations on data, or both. A trading agent is appointed by database systems to managing the services to be provided. The trading agent will be involved in special functions such as schema translation and shared data and/or operations on multiple database systems. The prototype implementation of the loosely integrated heterogeneous database system (involving two different database systems) is now running on a network of Sun workstations.

Key Words: *Heterogeneous databases, Traders in open distributed processing, Distributed systems, Client / server model.*

1 Introduction

Two approaches are commonly used in integrating heterogeneous database systems. The first is called the *unified schema* approach [12], and the second the *multidatabase* approach [13]. Both approaches acknowledge that there are a number of database systems in existence, and the design task involves integrating them into one virtual database system. The unified schema approach starts from individual local databases and translates each participating local conceptual database schema into a common intermediate database schema (a canonical representation). It then integrates each intermediate schema into a global conceptual schema [3]. Sometimes the local external schemas are considered for integrating rather than local conceptual schemas, since it may not be desirable to integrate the entire local conceptual schema in the integrated database.

However, the above approach essentially returns to centralisation by re-integrating the decentralised data into a "composite database." A global/virtual schema is used to describe the information in the databases being composed. Database access and manipulation operations are then mediated through this new conceptual schema. This type of integration can be called *tight integration.*

The process of integrating existing databases forces control over the local database structures (both conceptually and physically) to be ceded to some central authority. The users of the existing databases may have expended considerable resources in developing their databases and may be reluctant to lose control of them.

The multidatabase approach has no single integrated schema. The shared data is represented either as the actual local conceptual or external schema definition.

In many applications, local databases are not willing to change their own structure or give up control over their data, yet they are willing to share certain information with other database systems. Also, centralising all the local databases into a global schema may be too expensive or even not necessary. The following two application examples.

- A hospital may wish to share a subset of the information about its operating theatre waiting list to some external agent. It is unlikely it would wish to expose other confidential information.
- Two companies are developing a product that involves the use of part of their own database information. It would be undesirable for either company to divulge rest of their respective databases.

The solution to the above problems is to let individual database system have the full control over their own databases, yet let them decide what portion of the database is going to be shared [13]. No centralised global schema is enforced. We call this type of integration *loose integration.*

The key issues behind loose database integration are local database autonomy and information sharing [7]. There has been a lot of research on integrating distributed heterogeneous database systems, such as Multidatabase [14], Mermaid [15], InterBase [4], Dataplex [5], Remote-Exchange [10], Pegasus [1], and DIRECT [11]. However, all the existing approaches use *passive* information sharing. That is, they let some authority (e.g., the global schema) decide what is the content and format of information shared. In this paper we present an integration approach that allows the existing database systems to maintain their autonomy, yet through their willingness, provides a substantial degree of information sharing. The main dif-

ference between our proposal and existing approaches is the modes of information sharing. Our approach uses *active* information sharing. That is, we let the participating database systems decide the content and format of information to be shared.

2 The trader

2.1 Description of the trader

A *trader* is a third-party object that links clients and servers in a distributed system [9]. By using a trader, servers can advertise (export) their service offers and clients can get (import) information about one or more exported service offers that match some objectives [6] [2]. Traders have been a subject of international standardisation for some time. The best known and ongoing standardisation work is the ISO Open Distributed Processing project [9].

We use a trader to manage the shared information among participating databases. If a database system is willing to share part of its information with other database systems, it *exports* that willingness as a *service offer* to the trader. The following three types of service offers are defined:

- *Data.* Each database system has a collection of data that might be of interest to other database systems. This information can be exported to the trader as a service offer that can then be accessed by others. This type of service offers is designed for direct data sharing.
- *Operation.* A database system may not wish to shared its data directly with other database systems. In this case, an operation can be created and exported to the trader for indirectly sharing this critical data.
- *Object.* An object contains a piece of data and the operations that manipulate the information. It is essentially the combination of the two previous types of service offers.

Any local database willing to share its information has to export relevant service offers to the trader. Application programs (clients) wishing to make use of the shared information have to *import* such service offers from the trader and then access the database(s) concerned.

There are two forms of trading:

- *Direct trading.* The database systems export their service offers directly to the trader. The clients import these service offers from the trader and access the relevant database systems directly.
- *Indirect trading.* The database systems appoint a *trading agent* to manage the service offers. The trading agent is then responsible for exporting the service offer to the trader. After importing the service offer, the clients call the trading agent and all accesses to the database systems have to go through the trading agent.

Figure 1 depicts the direct trading process involving one database system only. In this figure, the database system (DB) exports its service offer to the trader; the client imports the service offer from the trader and the trader returns the offer to the client. The returned offer contains information such as the description of the service and the address of the database system that provides the service; then the client calls the database system directly and the result is returned.

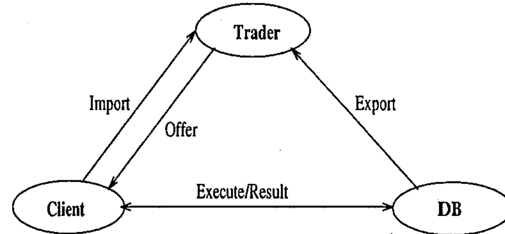

Figure 1: Trading process: direct trading

Direct trading may not be appropriate in many cases. For example, if some schema translation is needed, or if a service offer involves accessing multiple database systems, then the indirect trading may be used. Figure 2 depicts the indirect trading process that involves two database systems. In this figure, both database systems (DB1 and DB2) are willing to share their information in one service offer. They then appoint an agent and the agent exports the service offer to the trader. The client imports the service offer from the trader and is given the agent's address (and the description of the service, of cause). It then calls the agent and obtains the service.

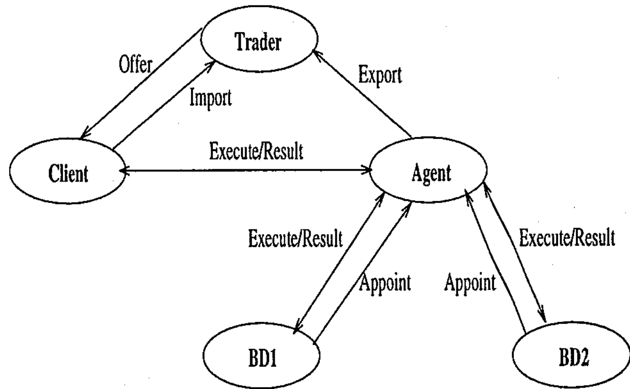

Figure 2: Trading process: indirect trading

The following questions must be properly answered in order to design such a trader systems:

- Trading operations. How do database systems, the trader, and trading agents interact with each other? How do users interact with the trader, trading agents, and database systems?
- Trading context management. All services must be registered by the trader before they can be shared. The total set of service offers managed by a trader is called the *trading context* of the trader. The question is, how are these service offers stored, accessed, and managed?

- An agent is used to provide services that need special treatment, such as a shared service involving two or more database systems. How do these database systems appoint the agent for building a common service for them?
- Is it feasible to build such a trader?

These questions will be addressed in the following sections.

2.2 Service offer data structures

A service offer is actually a set of capabilities that are provided by a server and are to be used by clients. We have defined three types of service offers, namely, data, operations, and objects. In order to describe these service offers in more detail, the following data structures are defined:

- Tables. Tables are the basic unit used in the trader to describe shared information. A database system willing to share its data with other systems must use the following data structure to describe it:

```
/* three types of service offers */
#define TAB_TYPE 1
#define TRA_TYPE 2
#define OBJ_TYPE 3

/* one attribute */
typedef struct AnAttr {
  char * aName;        /* attribute name */
  char * aType;        /* attribute type */
  int aLen;            /* maximum length */
} AnAttr;

/* one table */
typedef struct TableStruct {
  char *tableName;        /* table's name */
  AnAttr attrs[MAXATTR];/* attributes */
  int degree; /* number of attributes */
  int cardinality;  /* number of rows */
} TableStruct;
```

The TableStruct data structure defines the name, attributes, type of each attribute, and the size of the table.

- Operations. Another basic unit for describing shared information is an operation (procedure). An operation is defined as follows:

```
/* one procedure (operation) */
typedef struct ProcStruct {
  char * procName;      /* procedure name*/
  int noOfParameters;/*# of parameters*/
  char ** params;       /* parameters     */
  char ** paramTypes;/*types of params*/
} ProcStruct;
```

The ProcStruct data structure defines the name and the parameters (names, types, and number of parameters) of an operation.

- Shared data. A database system can offer a few tables for information sharing. The data structure for such data is as follows:

```
/* shared tables */
typedef struct DataStruct {
  char * dataName;      /* name of data */
  int noOfTables;         /* # of tables */
  TableStrcut TB[MAXTBS];     /* tables */
} DataStruct;
```

- Shared operations. Shared operations are offered by database systems that want to share their data indirectly. The data structure for such operations is as follows:

```
/* shared operations */
typedef struct TransStruct {
  char * transName; /*operations name */
  int noOfProcs;     /*# of procedures */
  ProcStruct PR[MAXPRS];/* procedures */
} TransStruct;
```

- An offer. An offer can be the form of shared data (a set of tables), shared operations (a set of operations), or a shared object (the combination of shared data and operations). The data structure of an offer is as follows:

```
/* an offer */
typedef struct AnOffer {
  char * offerName;     /* offer name    */
  int offerType;        /* offer type    */
  char * path;          /* context path */
  char * description;  /* offer descrip*/
  char * address;       /* service addrs*/
  DataStruct dName;     /* shared data   */
  TransStruct tName;   /* operation     */
} AnOffer;
```

If the offerType = TAB_TYPE, then the offer is to share a set of tables. The dName will contain the detail of these tables, and the tName will be empty. Similarly, if offerType = TRA_TYPE, then the offer is to share a set of operations. The tName will contain the detail of these operations and the dName will be empty. If offerType = OBJ_TYPE, then the offer is to share a set of tables and a set of operations on these tables. In that case, dName will contain the detail of shared tables and the tName will contain the detail of the operations.

2.3 Converting to/from service offer data structures

An exporter wishing to export its services has to use the service offer data structure described in Section 2.2 for describing its offers. If the local database system does not use the service offer data structure in its local definition, a translation is then needed. This can be done through the appointment of an agent. The agent is then responsible for converting the shared portion of the local schema into the service offer data structure. It is also responsible for converting the calls of the shared data structure into the local schema.

An importer imports from the trader an offer described in the service offer structure. It then uses this data structure to access the shared information. If the data structure of the service offer is different with

the importer's data structure, a translation is again needed. We assume that the importer will provide the mechanism for translating the service offer's data structure into its local data structure.

3 The trading operations

Three sets of operations have been designed and can be used by exporters, importers and the trader. One common feature of all these operations is the errors returned from the operations. We have defined a global data structure called opErrors that contains most of the possible errors from an operation.

Each operation will return one of the following values when the operation terminates:

```
#define OP_OK 0
#define OP_ER 1
```

The global data structure opErrors contains the error descriptions when the operation returns an OP_ER.

3.1 Exporter operations

The basic operation needed by an exporter is to export its service offers to the trader. The exporter may also need to withdraw a service offer if it does not want to provide the service any more. If some conditions have been changed, the exporter may also want to change the service offer accordingly. It is also necessary for the exporter to obtain some information about an existing service offer placed in the trading context by itself or by other exporters. The following operations have been designed and can be used by an exporter:

- Export. This operation is used by an exporter to export its service offer to the trader. The trader has a well-known address and is hidden inside the operation. This operation has the following format:

```
int export(offer, createOpt)
AnOffer offer;
int createOpt;
```

  Before calling this operation, the exporter has to fill in the offer structure with proper information, such as the type of the offer, the description of the offer, and the details of the offer. Most importantly, it has to give the preferred context path name for the offer to be stored within the trading context. When the offer reaches to the trader, the trader will use this path name to store the offer into the trading context. The structure of the trading context is discussed in Section 4. The exporter also needs to specify its address in the offer structure.

- Withdraw. This operation is used by an exporter to withdraw an offer that it placed at the trader at some prior time. The operation has the following format:

```
int withdraw(offerName, path)
char * offerName;
char * path;
```

  Both the offerName and path must be the same as used by the exporter in the export operation. The path is used by the trader to find the service offer in the trading context and the offerName is used to confirm the name of the offer to be deleted from the trading context.

- Replace. This operation is used by an exporter to replace an offer that it placed at the trader previously. The operation has the following format:

```
int replace(prevPath, prevName, offer)
char *prevPath;
char *prevName;
AnOffer offer;
```

  The prevPath and prevName contain the context path and the offer name that the exporter used during the export operation for placing the particular offer. This operation can change all the details of the service offer such as the name, the description, the data and/or operations, and so on. Even the context path name can be changed. In that case, the old service offer will be deleted from the trading context and then a new service offer will be stored by using the new context path and the new offer name contained in the offer data structure.

- Describe. This operation is used by an exporter to get some information about a particular offer. The operation has the following format:

```
int describe(path, offerName,
             offerType, description);
char * path;
char * offerName;
int  * offerType;
char * description;
```

  The path is the context path name of the service offer that the exporter wants to know about. The operation then returns the type of the service offer and the descriptions about the service.

3.2 Importer operations

The basic operation needed by an importer is to import a service offer from the trader. In many cases, it is also necessary for an importer to browse through the trading context and to obtain some descriptions about a particular service offer. The following operations have been designed and can be used by an importer:

- Import. This operation lets the importer import a service offer from the trader. The importer has to be aware of the context path and the offer's name of the service offer before it can invoke the import operation. The operation has the following format:

```
int import(path, offerName, offer)
char * path;
char * offerName;
AnOffer * offer;
```

After receives the import request, the trader will search its context space for the given path and the offer's name. If the search is successful, the selected service offer will be stored in the offer and returned to the importer.

- Describe. This operation is the same as the describe operation for an exporter.
- List. The describe operation only returns the descriptions of a particular service offer. Sometimes it is necessary to know about a set of offers that provide the same services. This operation is used to return the types, context path names, and descriptions of a set of service offers. The operation has the following format:

```
int list(offerName, noOfOffers,
         offerTypes, paths,
         descriptions)
char * offerName;           /* offer name */
int  * noOfOffers;  /* # offers found */
int  * offerTypes[];     /* offer types */
char * paths[];        /* context paths */
char * descriptions[];/*offer descrpt */
```

The operation takes the offerName as the input parameter and asks the trader to find out all service offers with the same offerName. If the operation is successful, the noOfOffers contains the number of service offers that match the offerName and the offerTypes, paths and descriptions contain the types, context path names, and descriptions of each of these service offers.

3.3 Management operations

All the management operations are used by the trader to manage the trading context. These operations allow the trader to add in new service offers, to delete or modify existing offers, to find the description about a specific offer, and to browse through the trading context. These operations can be (and actually have been) used by a utility tool that talks to the trader directly for managing the trading context. We will not go into the details of these operations since the functions of these operations are similar to the operations provided to the exporters and importers. The only difference is that these operations are now performed locally within the trader, or between the trader and the management utility.

4 Trading context management

The trading context of the trader consists of all the service offers registered with the trader. Associated with each service offer is the information that describes the service. That may include, for example, the name / address of the agent or database system that offers such service, the type of the offer, and a brief description of the service (e.g., a brief introduction of how to use the service),

Service offers of the trading context are managed by the trader as a tree structure, similar to the UNIX directory structure. Figure 3 describes an example of the trading context.

When exporting a service offer, the agent or the database system should specify the intended path of

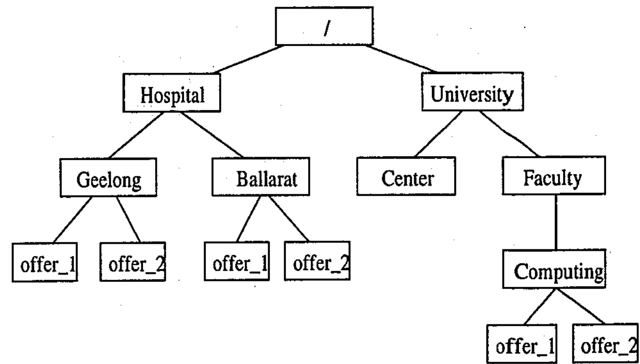

Figure 3: Trading context

the trading context and the offer's name for the service offer. The trader then uses the path and the offer's name to record and locate the service offer within the trading context.

The trading context also contains a linked list for offerName fields. This link list is used for obtaining descriptions and other information about a group of offers that have the same offerName.

The actual trading context uses only one set of context path names and two sets of pointers. One set of the pointers is used for constructing the context tree structure and the other set of pointers is used for constructing the linked list of offerNames.

The client can browse through the trading context and then find the desired service offer (through the list operation, for instance). The client can also obtain a more detailed description from the database or it appointed agent.

5 The trading agents

A trading agent integrates services involving two or more database systems, or provides some special management for the service offer. If two databases are heterogeneous, translators for transforming local schemas into a virtual schema may be necessary. The virtual schema is determined by the agent and may be negotiated by the participating database systems. Although we only use a label "Appoint" in Figure 2 to represent the process of trading agent appointment, the process would actually involves several communication steps among the participating database systems and the trading agent.

In our prototype implementation, we have not implemented a standard facility for trading agent appointment and for negotiations among participating database systems. At this moment trading agents are hand-written programs (with the help of a precompiler, see Section 6 for details) that perform the functions such as accepting local offers from participating database systems, format translating between local offers and the virtual offer, exporting the virtual offer to the trader, and processing requests from clients. The local offers of participating database systems are decided before the program of a trading agent is written.

6 Prototype implementation

Currently the trader has been implemented on a set of networked Sun workstations. Two database systems are used in the prototype: an Oracle database system and a Mini SQL database engine [8]. An Ingres database system and a POET object-oriented database system have been planned to be added into the system soon.

6.1 Architecture

Figure 4 depicts the architecture of our loosely integrated heterogeneous database system. Currently two database systems (Oracle and Mini SQL) are running on two separate Sun workstations. The trader, trading agents, and user application programs can be executed on any Sun workstations.

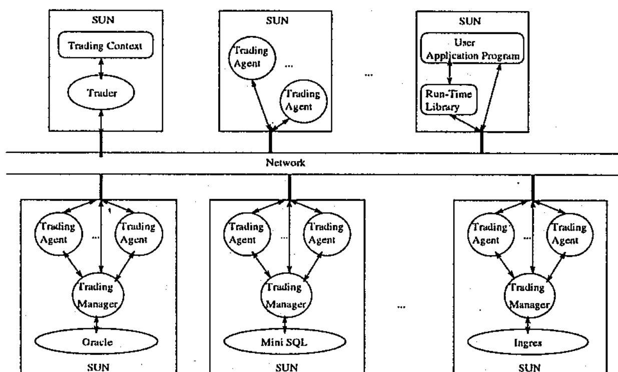

Figure 4: Architecture of the loosely integrated system

A *trading manager* is built on every participating database system for performing all common tasks of trading preparation and management. It is responsible for such tasks as checking the validity of trading requests, forming local offers, executing the service, and returning request results. By using the trading managers, we can reduce the duplicating part of each trading agent and concentrate on the work of the trading agent such as schema translation and combination. Of cause, it also adds one more level of interprocess communication.

One may argue that having one trading agent for each offer is a waste of resources. It is true to some extend, but it is also the mechanism that provides diversity of service requests and loose integration. These agents are application-oriented. For instance, if a special application requires some information from two or more heterogeneous database systems, the trading agent provides an easy mechanism for loose integration. A service offer can be deleted from the trading context if it is no longer needed.

The rapid prototyping tool described in [16] is used in the prototype implementation.

The trader is implemented as a server. It provides export and import operations through remote procedure calls (RPCs). The trader also uses a well-known port for accepting calls from clients. The context space is currently located in the main memory because of the small size. It should be implemented on a file systems if the space required becomes large enough.

The trading agent acts as both a server and a client. To the user program, a trading agent is a server because it provides services that the user program has imported from the trader. But to a trader program, a trading agent is a client because it exports service offers to the trader. The interprocess communication between trading agents and the trading manager is through sockets.

6.2 Offer definition files

We use an *offer definition file* (ODF) to define an offer and then through a precompiler called *offer frame generator*, the source files of the offer defined in the ODF will be generated. Currently only C source code is generated. Listing 1 shows the syntax of an offer definition file.

Listing 1. Offer definition file syntax

```
ODF      ::= BEGIN
               INCS
               OFFER
               [ TABLES ]
               [ PROCS ]
             END
INCS     ::= [ INC ]
INC      ::= Include: filename ;
OFFER    ::= Offer Name: variable ;
               Offer Path: string ;
               Offer Description: variable ;
               Offer Address: HOST, PORT ;
HOST     ::= string
PORT     ::= integer
TABLES   ::= Table: string from [variable];
              TBS ;
              End Table ;
TBS      ::= TB { TB }
TB       ::= Name: string ; ATTRS
ATTRS    ::= { ATTR }
ATTR     ::= Attrib: declarator ;
PROCS    ::= Procedures: string; OPS ;
             End Procedures ;
OPS      ::= OP { OP }
OP       ::= Name: string ; PARAMS
PARAMS   ::= { PARAM }
PARAM    ::= Param: CLASS: declarator ;
CLASS    ::= in | out | in_out
```

Most of the descriptions of Listing 1 are self-explanatory. We use a modified BNF to denote the syntax of definition files, where [x] means that x can appear 1 to many times and {x} means that x can appear 0 to many times. The "variable", "integer", "string", and "declarator" have the same meanings as in the C programming language. Comments are allowed in the definition file. They are defined the same as in the C programming language (using /* and */). The semantics of an ODF file will be made clear in Section 6.4.

6.3 Offer frame generator

After a programmer sends an offer definition file to the offer frame generator, the generator first does syntax checking. If no errors are found, several program source files are generated. These generated files can be used by programs that use the service offer, such as the trading agent, the trading manager, and the user program. A *makefile* is also generated for testing the

service offer by using the generated driver programs. That is, when using the make utility, the executable files for the trading agent, the trading manager, and the user program will be generated. Figure 5 shows the input and output of the offer frame generator. By default, source code for the trading agent is always generated. If the user does not want trading agent to be generated, the source code for trading manager and user application program will be a little different (for instance, the trading manager in this case will include operations previously located in the trading agent).

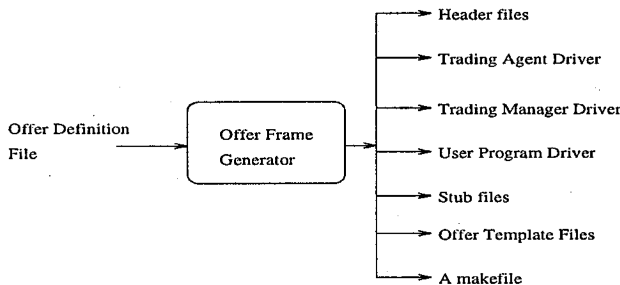

Figure 5: Input and output of the offer frame generator

6.4 Example applications

We use a simple example to show the application of the tools and the idea of loosely integrating heterogeneous database systems. Suppose we have a database about off-campus student records and it is stored in an Oracle database. The table definition is assumed to be as follows (the database may contain other tables):

```
OFFCAMPUS(sid integer, sname char[20],
          street char[20], suburb char[20],
          city char[10], country char[12],
          phone char[16], major, ...)
```

Assume that we have decided to share those student records that have `major = 'Computing'` in some applications. We may want to define an offer as follows:

```
/* Offer from off-campus student database */
BEGIN
  Include:      offCampus.sql
  Offer Name: ofcs;
  Offer Path: /University/Faculty/Computing;
  Offer Description:
   "A table for off-campus students majoring
    in Computing. In Oracle database. An
    operation to change student address";
  Offer Address: frodo.deakin.edu.au, 6500;

  Tables:             StudentTable;
    Name: Student from OFFCAMPUS;
      Attrib: int sid;
      Attrib: char sname[20];
      Attrib: char street[20];
      Attrib: char suburb[20];
      Attrib: char city[10];
      Attrib: char country[12];
      Attrib: char phone[16];
  End Tables;

  Procedures:
    Name:  changeAddress;
      Param:  in: integer sid;
      Param:  in: char street[20];
      Param:  in: char suburb[20];
      Param:  in: char city[10];
      Param:  in: char country[12];
  End Procedures;
END
```

The offer has one table which shows the details of students that are majored in "Computing", and an operation that changes a student's address. The trading agent is to be executed on machine `frodo.deakin.edu.au` and is to use socket port 6500. The offer is going to be stored in the trading context path `/University/Faculty/Computing` under the name of `ofcs`. The table of the offer is created from a database definition file named `offCampus.sql`, which contains a `CREATE TABLE` statement for the `OFFCAMPUS` table. After we send this file to the offer frame generator, the following files will be generated:

```
ofcs.h          Header file, must be included
                by trading agent, user
                program, and trading manager.
ofcsTA.c         Trading agent driver file
ofcsTAStub.c     Trading agent stub file
ofcsTAOps.c      Framework of trading agent
ofcsTM.c         Trading manager driver file
ofcsTMStub.c     Trading manager stub file
ofcsTMOps.c      Framework of trading manager
ofcsAP.c         User application driver file
ofcsAPStub.c     User application stub file
makefile         make file
```

All these files use the data structure defined in Section 2.2. After using the **make** utility, three executables will be generated:

```
ofcsTA       Trading agent program
ofcsTM       Trading manager program
ofcsAP       User application program
```

The `ofcsTMOps.c` file contains such operations as security checking, view creating, offer creating, offer executing, and so on. The `ofcsTAOps.c` file contains such operations as offer exporting, format translating, and so on. Note that some of the operations (such as the `changeAddress` operation defined in the definition file) are not fully defined in these files. Instead, only frameworks (dummy procedures) are defined for such operations. Their details are to be programmed by the programmer.

Now suppose we have another ISAM database containing on-campus student records and is defined as follows:

```
ONCAMPUS(sid integer, studentName char[30],
         address char[40], phone char[10])
         major, ...)
```

Assume that both databases want to share their student records with "Computing" major in some applications. We can define the following offer to accommodate the differences of the two databases:

```
/* Both */
BEGIN
  Include:     offCampus.sql
  Include:     onCampus.sql
  Offer Name:  allcs;
  Offer Path:  /University/Faculty/Computing;
  Offer Description:
    "A table for all students majoring in
    Computing. In both Oracle and ISAM DBs;
  Offer Address:baragund.deakin.edu.au,6700;

  Tables:                StudentTable;
    Name:  Student from OFFCAMPUS, ONCAMPUS;
      Attrib: int sid;
      Attrib: char name[30];
      Attrib: char address[65];
      Attrib: char phone[16];
  End Tables;
END
```

The offer contains only one table. It is going to be executed on the `baragund.deakin.edu.au` machine and is to use socket port 6700. The table is created from two database definition files named `offCampus.sql` and `onCampus.sql`. These two files contain the definitions for `OFFCAMPUS` and `ONCAMPUS` tables, respectively. The offer is going to be stored in the trading context path `/University/Faculty/Computing` under the name of `allcs`.

The trading agent in this case will be responsible of operations including forming the virtual table from the two tables, directing user requests to different database trading managers, accepting results from trading managers, and converting the results into the virtual table format. Note that because the `address` attribute in the global table combines a few attributes from the student table of the off-campus database, any update to this field will then be prohibited by the trading agent.

7 Remarks

The design and prototype implementation of a loosely integrated heterogeneous database system is described in this paper. The main contribution of this paper is the introduction of the active participation of information sharing by the database systems and the introduction of data and operation sharing. It also shows that building such a system that loosely integrates heterogeneous database systems is possible.

References

[1] R. Ahmed. The pegasus heterogeneous multidatabase system. *Computer*, 24(12):19–27, December 1991.

[2] M. Bearman. ODP trader. In *Open Distributed Processing II, IFIP Transaction C-20*, pages 37–51. Elsevier Science B. V., North-Holland, 1994.

[3] Z. Brzezinski, J. Getta, J. Rybnik, and W. Stepniewski. Unibase: An integrated access to database. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 388–400, Singapore, 1984.

[4] O. A. Bukhres, J. Chen, W. Du, and A. K. Elmagarmid. InterBase: An execution environment for heterogeneous software systems. *Computer*, 26(8):57–69, August 1993.

[5] C. W. Chung. Dataplex: An access to heterogeneous distributed databases. *Communications of the ACM*, 33(1):70–80, January 1990.

[6] A. Goscinski and M. Bearman. Resource management in large distributed systems. *Operating System Review*, October 1990.

[7] D. Heimbigner and D. Mcleod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[8] D. J. Hughes. *Mini SQL: A Lightweight Database Engine.* Huges Technologies P/L, Australia, http://Hughes.com.au, January 1996.

[9] ISO/IEC. *Working Document - ODP Trading Function.* ISO/IEC JTC1/SC21 N8409, 1994.

[10] D. McLeod. The remote-exchange approach to semantic heterogeneity in federated database systems. In *Proceedings of the 2nd Far-East Workshop on Futhre Database Systems*, pages 38–43, April 1992.

[11] U. Merz and R. King. DIRECT: A query facility for multiple databases. *ACM Transactions on Information Systems*, 14(4):339–359, October 1994.

[12] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, Englewoods Cliffs, New Jersey, 1991.

[13] S. Ram. Heterogeneous distributed database systems. *Computer*, 24(12):7–10, December 1991.

[14] J. M. Smith, P. A. Bernstein, U. Dayal, N. Goodman, T. Landers, K. Lin, and E. Wong. MULTIBASE: Integrating heterogeneous distributed database systems. In *Proceedings of the American National Computer Conference*, pages 487–499, MAY 1981.

[15] M. Templeton, E. Lund, and P. Ward. Pragmatics of access control in mermaid. *IEEE Data Engineering Bulletin*, 10(3):33–38, 1987.

[16] W. Zhou. A rapid prototyping system for distributed information system applications. *The Journal of Systems and Software*, 24(1):3–29, 1994.