Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Remote Database Queries in Open Distributed Systems

Gomer Thomas &
Rob van der Linden

Abstract

Both the remote database access (RDA) paradigm, based on remote query language access to data, and the remote procedure call (RPC) paradigm, based on remote calls to predefined procedures, have become increasingly important in recent years. Each has unique advantages and disadvantages. Unfortunately, they are currently supported by different protocols and different distributed computing infrastructure. There is a great need to unify these paradigms, so that a single set of infrastructure can support mixed paradigm programming and allow application developers to enjoy their combined advantages. This paper uses the object based concepts of the ANSA computational model for open distributed computing to show how these two paradigms may be aligned. The resulting model can be applied to current relational databases and SQL, and it suggests an architecturally sound approach to implementing remote SQL access in an ODP environment. It also lays a solid foundation for the remote access capabilities which will be needed by the object databases of the future.

APM.1163.00.01 **Draft** 25 February 1994

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Remote Database Queries in Open Distributed Systems

Remote Database Queries in Open Distributed Systems

Gomer Thomas & Rob van der Linden

APM.1163.00.01

25 February 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Two Distributed Computing Worlds

Two distinct distributed computing paradigms have become increasingly important in recent years: the remote database access (RDA) paradigm, based on remote query language access to data, and the remote procedure call (RPC) paradigm, based on remote calls to predefined procedures. (The distributed object model, with its remote invocations of operations on objects, can be viewed as essentially an RPC model, with powerful abstractions for structuring the RPC interactions.)

This chapter describes the two paradigms, describes the advantages and disadvantages of each, and explains the compelling practical motivation for a conceptual model which unifies them.

1.1 Remote Database Access World

The key feature of the RDA paradigm is that applications access remote databases through query language calls, e.g., SQL, in much the same way as they would access local databases. One important consequence of this is that distributed applications can use the same advanced development tools as are used for centralized applications -- report writers, 4GLs, spreadsheet interfaces, graphing interfaces, interactive browsing tools, etc.

A commonly used API is embedded SQL, in which SQL statements are embedded in program code. Typically a preprocessor converts the embedded SQL statements into calls to library procedures which handle message encoding/decoding and communications with the remote server(s). Another common type of API is the call level API, where the program calls the encode/decode/communications procedures explicitly. Because the numbers and types of arguments and results for an SQL statement are determined by the text of the statement itself, the procedures use some form of dynamic parameter definition. Open APIs for remote database access include the ANSI and ISO SQL standards for embedded SQL [ANSI-SQL 92] [ISO-SQL 92], the X/Open SQL Call Level Interface (CLI) specification [CLI 92], and the Microsoft ODBC call level specification [ODBC 92].

Open protocols for remote database access include the ISO RDA/SQL standard [RDA-1 92] [RDA-2 92], the IBM DRDA specification [DRDA], and the SQL Access Group RDA for TCP/IP specification [RDA-TCP 93]. The first of these is dependent on the OSI protocol stack, the second is dependent on the SNA LU6.2 protocol, and the third is dependent on the TCP/IP protocol. All provide some means for including descriptions of the types of arguments and results in messages, as well as the values.

Common vendor extensions to the basic RDA paradigm include distributed query processing (the ability to reference data from multiple data sources within a single query language statement), distributed transaction management (the ability to include operations at multiple data sources within a single global transaction), and stored procedures.

1.2 Remote Procedure Call World

The key feature of the RPC paradigm is that applications invoke remote services by invoking procedures, in much the same way as they would invoke local procedures. Usually the signature of the procedure is defined by use of an interface definition language (IDL), and the IDL description is available to both client and server at compile time. Typically preprocessors utilize the IDL description to generate client and server stubs to be linked with the client and server application code.

Distributed object-oriented environments, such as that specified by OMG [OMG 90] [CORBA 91], can be viewed as a special case of the RPC paradigm in which the object concept is used to structure the set of available procedures and the ways they are used.

Open APIs for remote procedure calls include the OSF DCE RPC specifications [DCE 92] and the OMG CORBA specification [CORBA 91]. Open protocols include the OSF DCE RPC specifications [DCE 92] and the ISO RO standard [RO-1 88] [RO-2 88].

The emerging ISO ODP (Open Distributed Processing) standard [RO-1 88] provides a reference model for distributed processing. It is an object-oriented framework, in which the invocations of operations on objects can be viewed as remote procedure calls.

A useful extension to the basic RPC paradigm is so-called transactional RPC, in which a conversation “handle” is passed between client and server, so the server can maintain interaction context between procedure invocations.

1.3 The Need for Unification

These two paradigms each have their unique advantages and disadvantages.

RDA provides great flexibility. With remote query language access interrelated data items can be selected and combined to meet the precise needs of diverse client applications, without the need for custom programming at the server. This can be very useful when application needs change rapidly as a result of new business opportunities or process re-eingineering. It is especially useful when the database is maintained by a different organization from that responsible for the applications accessing the database. Making new procedures available at the server in this circumstance can be very costly and time consuming.

On the other hand, RPC allows client applications to operate at a higher level of abstraction, which supports more convenient client application programming, better protection for semantic integrity of the database during updates, and lower communications overhead. For example, a remote application might invoke a single procedure to add N units of item X to a shipment going out to customer Y. The procedure would update the database of available inventory, add the items to the pick list for the shipping department, add the items to the invoice for the accounting department, and take care of any other necessary sub-tasks in connection with the overall task. The RPC is much more efficient for the developers of remote applications than having to program each data access individually. Moreover, it does not require them to understand all the details of the data structures in the relevant databases. The RPC can protect data consistency by performing all necessary checks and updating all interrelated data items, while individual client

application programmers using remote SQL may or may not do so in all cases. The RPC requires only a single exchange of messages between client and server, rather than the multiple exchanges which would be required to carry out the same tasks via individual remote SQL calls.

RPC can also be used to invoke tasks or services which do not involve data.

So-called “stored database procedures” can be used to provide the advantages of the RPC paradigm in an RDA environment, but they are not a totally satisfactory substitute, as shown by the burgeoning use of RPC environments even for database applications. One problem is that stored procedures are a much newer concept than RPCs and are not yet standardized. Another problem is that many applications need to invoke non-database operations or processes as well as database operations.

It may be possible to use triggers [DATE 83] to protect semantic integrity of data in an RDA context, but so far no systematic methodology for doing so has been demonstrated for large, production applications.

Unfortunately, the RDA and RPC paradigms have given rise to distinct and incompatible application programming interfaces (APIs), protocols, and underlying infrastructure. This creates a major problem for applications which need to realize the combined advantages of the two paradigms. Even purchasing and installing two sets of infrastructure may not solve the problem. An application may need to make interspersed RDA calls and RPC calls to the same server and have them all treated as part of a common transaction thread. With current products a distributed transaction monitor would be needed to achieve this, even though there is only a single client and a single server involved in the interaction!

In order to enable mixed paradigm programming, it is necessary to lay a conceptual foundation for a unified RDA/RPC computing infrastructure.

At a simple operational level, the barriers to implementing RDA invocations over an RPC infrastructure are:

- RPC environments typically assume static pre-definition of operations, but the numbers and types of parameters of an RDA invocation are defined dynamically.
- RPC environments typically do not provide convenient mechanisms for the server to maintain interaction context between procedure calls from the same client.
- RPC environments typically do not provide convenient mechanisms for an operation to return very large, variable quantities of data.

At a more fundamental level, the RDA and RPC paradigms are based on different, incompatible conceptual frameworks for how applications interact in a distributed environment.

This paper proposes an object based conceptual model for query language access to databases which provides a basis for unification of query language and procedural access to data. The model does not assume that the database itself is based on an object model, nor any other particular data model. In the near term, the model has a natural interpretation in the context of relational databases and SQL, and it suggests an architecturally “clean” approach to implementing RDA over an RPC infrastructure. In the longer term, it suggests a very flexible and powerful model for implementing remote query language access to object databases in a distributed object system.

2 Unifying the Two Worlds

The purpose of this section is to describe an object-oriented model of remote interaction with databases which unifies the RPC and RDA paradigms. No particular data model is assumed for the databases themselves. They may be object-oriented, relational or whatever.

This model grew out of efforts to incorporate remote query language access to databases into the ANSA framework. Therefore, it will be described in the language of the ANSA computational model [APM1001 93], an object-based model of client-server interaction in distributed systems.

The ANSA computational model makes a distinction between an object and an interface to an object. An object is a unit of program modularity having state and operations for initializing, accessing and updating that state. An interface is a view of an object as an abstract service, specified as a set of operations. The type of an interface is determined by the operations it supports. A single object may have multiple type of interfaces, and multiple instances of each type. Among other things, an interface may encapsulate the state of the interaction with a client, separate from the state of the underlying object.

In the ANSA computational model arguments and results of an operation are always references to interfaces. By slight abuse of terminology, we will often speak of arguments and results as references to objects, or sometimes just as objects. An operation may have multiple "terminations," distinguished by name. A common usage of this concept is to have one "normal" termination and one or more "abnormal" terminations. The different terminations may have different numbers and types of results.

2.1 Model of a Database

A database can be considered as a collection of sets of entities, where each set has a type associated with it, and all the entities in the set are of that type. Entities may be of arbitrarily complex structure. Entities in the same or different sets may or may not be interrelated by various kinds of semantic relationships. Entity types may or may not be related by subtype-supertype relationships. In a relational database the sets are the relational tables, and the entities are rows. In an object database the sets are object classes, and the entities are individual objects. The individual entities are typically created and deleted dynamically during the day to day functioning of the database, often at a great rate.

Certain data operations can be associated with the database itself, for example a join over multiple entity sets. Certain operations can be associated with individual entity sets, for example the selection of a particular subset of entities from the entity set, or the computation of an aggregate function over the entity set. Certain operations can be associated with individual entities, for example an update to the attributes of a single entity -- although in the presence of semantic integrity constraints between entities even a simple

update to an entity may need to be viewed as a set operation or a database operation.

Thus, one can consider the individual entities in the database to be objects. Following OMG [CORBA 91], attributes of entities can be modeled as pairs of operations set_ attr_*x* and retrieve_attr_*x* with the obvious semantics. The entity objects may have other operations as well, as they often would in the case of an object-oriented database.

The sets of entities can also be considered to be objects. Each set object consists of references to the entity objects in the set, together with certain set operations. In performing these set operations, the set objects may in turn invoke operations on the entity objects. For example, to delete all entity objects having a specified value for a specified attribute, the set object would invoke a retrieve operation for the given attribute on each entity object in the set to determine which objects qualify. Then it would request that the entities destroy themselves, and it would drop their references.

The database itself can be considered as an object. It consists of references to the entity sets in the database, together with certain database level operations, including query language operations and perhaps predefined procedures. These operations manipulate entities or sets of entities, in general involving entities of multiple types. In performing the database level operations, the database may in turn invoke operations on entity set objects and individual entity objects.

Figure 1 illustrates this model.

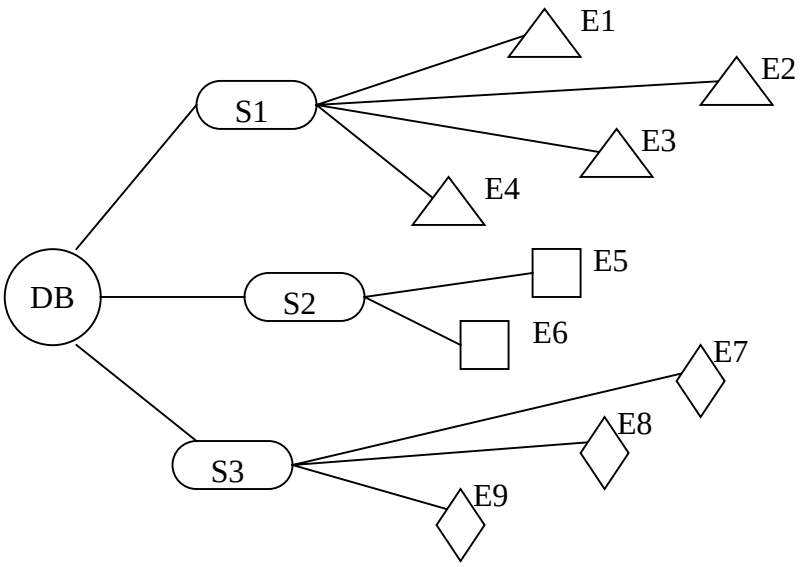

Figure 2.1: Conceptual Model of a Database

This is of course a conceptual model and may not reflect the actual implementation. In a relational database, for example, database operations are actually performed by means of highly optimized internal data manipulations, not by invoking operations on the tables and rows as independent objects.

This model makes no assumptions about the distribution of the entity objects, set objects or database object. They may all be at the same location, or they may be widely scattered.

The model extends easily to include the functionality normally associated with distributed or federated databases, by adding higher level “super-database” objects. A super-database consists of references to databases (possibly including other super-databases), together with operations which span databases. To carry out the super-database operations it may in turn invoke operations on the individual databases.

2.2 Client-Server Interaction Model for Databases

RDA protocols are typically connection-oriented. This reflects the fact that a database server must often maintain interaction context between operations. For example, when a client invokes multiple operations within a single transaction, the database must keep track of the uncommitted operations, or when a client executes a select statement and starts to fetch the resulting data, the database must keep track of the data which remains to be fetched.

Thus, a database should offer a “connection” interface, through which a client can request a logical connection and perhaps provide appropriate authentication information. When a connection is requested, the database should create a new instance of a query interface, and provide the client with a reference to that interface. The client then performs all query operations through that interface. When the client closes the connection, the interface instance created for the client is deleted, together with any state associated with it.

As will be seen, many of the operations invoked through such an interface are dynamically defined, in the sense that the numbers and types of the arguments and results are determined by the query statement itself. Thus, a query statement can be viewed as a specification of an operation, or as an operation constructor. The client application may or may not know the type of the operation in advance. For example, if the client application is a generic database browsing tool, the tool may get a query from a human user without the tool having any advance knowledge of the nature of the query. The database should provide a “describe” operation (analogous to the SQL DESCRIBE statement) for such situations. The input to the describe operation would be a query string. The output would be a type description, i.e., an object of *type* object-type. This is discussed in more detail in section 5.

2.3 Query Language Operations Model

We now present a model for query language operations within the framework of the proposed conceptual model of a database. The generic types of operations we consider are analogs to the different types of SQL data manipulation statements.

In describing result types we consider only the normal termination. In the event of any sort of error condition there would be an abnormal termination returning an error code.

2.3.1 Query Language Select Operations

Remote query language retrieval can be viewed as taking place in two phases. First there is a selection phase where the data manager generates the results of the query. Then there is an access phase where the client accesses the

results. The “select” statement can be viewed as defining a new object type, constructed from the existing types in the database, and defining a new set of objects of the new type, constructed from the existing objects in the database. Executing the selection statement causes the new set to be created and a reference to it to be returned to the client. The client can then access the query results by invoking any operation supported by the set object.

In the case of relational databases and SQL, the new set (a virtual table) is created by execution of a SELECT statement, or more precisely by execution of an OPEN statement on a cursor associated with the SELECT statement, and then the operations supported by the set object are:

- Retrieval of the individual objects (rows) in the set through FETCH statements
- Where there is an unambiguous one-to-one relationship between objects in the new set and objects in the original database from which they were constructed, update or deletion of the objects through positioned UPDATE and positioned DELETE statements.
- Destruction of the set through a CLOSE cursor operation.

In a more general object-oriented context one could envision a much more extensive collection of operations being supported by the new set and its component objects.

Taking an object-based view of the standard SQL statement types: The arguments to an SQL selection operation are the text of the selection statement and any parameters for which there are place-holders in the statement. The result is a reference to the set of objects created by the statement. The only argument to an SQL “fetch” operation is a reference to the “selected” set of objects. The result is an object of the type defined by the selection statement. The only argument to a “positioned delete” operation is a reference to the object to be deleted. There is no returned result. The arguments to a “positioned update” operation are a reference to the object to be updated and the new values for the attributes to be updated. There is no returned result.

2.3.2 Query Language Delete Operations

A searched delete operation deletes a subset of a set of objects, where the subset is determined by a selection clause. The arguments to the operation are the text of the delete statement and any parameters for which there are place-holders in the selection clause. There is no returned result.

The so-called “positioned delete” was discussed in section 2.3.1.

2.3.3 Query Language Update Operations

A searched update operation updates selected attributes of a subset of a set of objects, where the subset is determined by a selection clause. The arguments to the operation are the text of the update statement, the new values for the attributes to be updated, and any parameters for which there are place-holders in the selection clause. There is no returned result.

The so-called “positioned update” was discussed in section 2.3.1.

2.3.4 Query Language Insert Operations

The “values” form of a query language insert operation inserts a single object into a set. The arguments are a reference to the object to be inserted, and a reference to the set into which it is to be inserted. There is no returned result.

The “query specification” form of a query language insert operation consists of the selection operation of a query language retrieval, followed by the inclusion of the new set of objects into an existing set in the database. The arguments of the inclusion operation are a reference to the newly created set and a reference to the set into which it is to be included. There is no returned result.

3 Further Issues

This chapter discusses a number of further issues related to the proposed model.

3.1 Type Management

An important aid to ensuring successful interoperation in an open distributed processing system is type checking, preferably performed at the earliest stage possible. Often this is accomplished by making IDL definitions of interfaces available to both client and server at compile time, together with the use of "trading" [ODP-RM 93] at run time to make sure the definition used by the client is still consistent with the one used by the server. The OMG CORBA specification [CORBA 91] provides for dynamic creation of invocation requests to objects, but it seems to still envision that the actual operations are pre-defined, with interface definitions stored in the Interface Repository.

The purpose of this section is to address the type checking problem in the context of the model of section 4, where the operations themselves as well as their invocations are dynamically created through query langauge statements. The discussion is more or less independent of any particular API.

To support dynamic type checking at run time, we assume that there exist objects of *type* InterfaceType, and there exist objects of *type* OperationType. An object of *type* InterfaceType represents the type of an argument or a result. (If this terminology seems confusing, you can consider InterfaceType to be much the same as what is often called ObjectType. It's not quite the same, but close enough for this discussion.) An object of *type* OperationType represents the type of an operation. Essentially, an OperationType object contains the same information as an IDL description of an operation — the name of the operation, and the types of the arguments and results.

There are two steps at which type matching is needed:

- Formatting and interpreting messages, which requires either a knowledge of the operation type or a self-describing form of message encoding.
- Converting data formats between program and messages, which requires a knowledge of program variable types, as well as a knowledge of the operation type.

In the case of query language data access three cases arise:

1. The client program uses static queries; i.e. the queries to be issued by the program are all known at compile time. Note that this does not mean the queries are known to the database server at its compile time.

   In this case a compile time preprocessor can look at (a) the text of each query, (b) the program variable declarations, and (c) the database schema, and can ascertain the correct operation type for each query. Thus, both correct message formats and correct conversion of variables to messages

can be achieved. If desired, the client can also invoke a describe operation at the server at run time to guarantee that there has not been a schema change which changed the type of the operation.

2. The program uses dynamic queries, but the programmer knows enough about the dynamic queries to be issued that the types of the arguments and results are known to the programmer at compile time. (A common case of this in an SQL context is a situation where the SELECT clause of a select statement is known, but the WHERE clause is dynamically generated, with no parameters appearing in the WHERE clause.)

   In this case the programmer can provide a description of the operation type, and the compile time preprocessor can use this description to perform the necessary checks. If desired the describe operation can be invoked to check at run time that the type description given is correct.

3. The program uses dynamic queries, and the numbers and types of the arguments and results are unknown at compile time.

   In this case the program is pretty well forced to invoke a describe operation at run time to ascertain the type of the operation defined by the query statement. This can be used to ensure correct message formats, but usually cannot be used to ensure correct conversion of data between program and messages, since type information on program variables is rarely available at run time.

3.2 Trading for Query Language Services

"Trading" is an increasingly accepted mechanism for helping clients match up with services in distributed systems [ODP-RM 93]. The general idea is that there are one or more "traders" which serve as repositories of information about services available in the distributed system, and clients can get from a trader the information they need to invoke the service(s) they need, including the correct signatures of operations.

An important issue is how to handle trading for a database service, for example a bibliographic search service or a restaurant rating service. In order for a client to access a database service successfully, it is necessary for the client to know the logical data model for the information provided by the database server (structure and semantics), the view presented to the client (if different from the logical data model), the mapping between the logical data model and the view, and the command language or query language used to access the view.

One option is for clients to obtain this meta-information from a third party trader. Another option is to have a third party trader contain only some broad descriptive properties of the database service, say a collection of key words describing the general categories of information it contains. The client would them obtain the meta-information directly from the desired database server itself, or perhaps from a specialized data dictionary (which of course can be viewed as a specialized trader). In either case the "describe" operation mentioned earlier can be used to provide signature information on operations defined dynamically by query language statements.

3.3 Practical Implications of the Model

From a practical standpoint, there are several approaches one could take to supporting the RDA and RPC paradigms with a common body of infrastructure:

1. Define a single RPC operation (“query”) with a single input parameter whose type is byte string of indeterminate length (unbounded in theory, although in practice systems usually enforce some upper bound) and a single output parameter whose type is byte string of indeterminate length (also unbounded in theory). Then encode some standard RDA protocol within these byte strings. This essentially uses the RPC invocation mechanism as simply a transport protocol. The application programmer would not see this RPC call at all. A preprocessor and/or function library would convert the embedded query language statements or the function calls generated by the programmer into the byte strings for the operation invocation.

   This has the short-term advantage that it would be fairly easy to implement for vendors who already support the standard RDA protocol chosen, since they already have all the software for formatting messages, and they only have to shift to a different means for transporting the messages.

   On the other hand, it has long-term disadvantages. It requires one set of infrastructure to marshal and unmarshal arguments for ordinary RPC calls, and a separate set on top of that for query calls. This kind of clumsy arrangement often leads to later problems. Moreover, it does not lay a solid foundation for future extensions to object databases and query languages.

2. Follow the approach suggested by the model proposed in this paper.

   This solution is quite elegant and lends itself well to probable future needs of object databases and object query languages such as that proposed by the Object Database Management Group [CATTELL 93]. The required support for dynamically defined operations is likely to prove useful for other types of applications as well. It does not appear to be difficult to implement, as long as the client infrastructure supports dynamic invocation, as in the CORBA specification. (A prototyping effort is currently underway in the ANSA project to test this hypothesis.)

3. Define a collection of operations, corresponding perhaps to the functions in the CLI specification [CLI 92]. Some of these would have fixed definitions of input and/or output parameters. Some would still have to fall back on the byte string of indeterminate length for some parameters.

   This unfortunately combines most of the disadvantages of the first two approaches, without realizing any of the advantages.

4 Acknowledgements

The authors are indebted to many members of APM, especially Mike Beasley, Andrew Herbert, Yigal Hoffner, David Iggulden, Chris Mayers, Owen Rees, John Warne and Andrew Watson, for many insights gained from them during discussions on the topic of this paper.

References

[ANSI-SQL 92]

ANS X3.135-1992, *American National Standard for Information Systems - Database Language - SQL*, American National Standards Institute, October 1992.

[APM1001 93]

The ANSA Computational Model, Architecture Report APM.1001.01, APM Ltd., February 1993.

[CATTELL 93]

R.G.G. Cattell, *Object Database Management Standard: ODMG-93*, Morgan Kaufmann, 1993.

[CLI 92]

Data Management: SQL Call Level Interface (CLI), X/Open Snapshot, X/Open Company, Ltd, September 1992.

[CORBA 91]

The Common Object Request Broker: Architecture and Specification, Version1.1, Object Management Group, December 1991.

[DATE 83]

C.J. Date, *An Introduction to Database Systems*, Volume II, Addison-Wesley, 1983.

[DCE 92]

DCE Application Development Reference, OSF DCE Version 1.0, Revision 1.0, Update 1.0.1, Open Software Foundation, 1992.

[DRDA]

Distributed Relational Database Architecture Reference, IBM publication SC26-4651, International Business Machines.

[ISO-SQL 92]

ISO/IEC 9075: 1992, *Information Technologies - Database Language - SQL*, International Organization for Standardization, 1992.

[ODBC 92]

Programmer's Reference: Microsoft Open Database Connectivity Software Development Kit, Version 1.0, Microsoft Corporation, 1992.

[ODP-RM 93]

Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model, ISO/IEC JTC1/SC21/WG7 Committee Draft, June 1993.

[ODP-Tr 93]

Information Technology - Open Distributed Processing - ODP Trading Function, ISO/IEC JTC1/SC21/WG7 Working Draft, November 1993.

[OMG 90]

Object Management Architecture Guide, Version 1.0, Object Management Group, November 1990.

[RDA-1 92]

ISO/IEC 9579-1: 1992, *Information Technologies -- Open Systems Interconnection - Remote Database Access - Part 1: Generic Model, Service and Protocol*, editor's final draft, November 1992.

[RDA-2 92]

ISO/IEC 9579-2: 1992, *Information Technologies -- Open Systems Interconnection - Remote Database Access - Part 2: SQL Specialization*, editor's final draft, November 1992.

[RDA-TCP 93]

Remote Database Access for TCP/IP, Consortium Specification, published by X/Open Company, Ltd., on behalf of the SQL Access Group, May, 1993.

[RO-1 88]

ISO/IEC 9072-1, *Information Processing Systems - Text Communications - Remote Operations: Part 1, Model, Notation, and Service Definition*, International Organization for Standardization, 1988.

[RO-2 88]

ISO/IEC 9072-2, *Information Processing Systems - Text Communications - Remote Operations: Part 2, Protocol Specification*, International Organization for Standardization, 1988.