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

Data Management for an Enhanced Trader

Gomer Thomas, Mike Beasley, Yigal Hoffner

Abstract

As the world moves toward large, heterogeneous, federated, distributed computing systems, the problems of matching up diverse clients and servers for interoperation take on increasing importance. “Trading” is a partial solution, allowing clients to locate, at run time, servers which meet stated constraints on interface type and certain other types of properties. However, matching will need to take place at earlier phases in the application life cycle as well, and will need to be based on much more complex constraints involving properties such as charging & billing policies, security & privacy policies, operation semantics, performability guarantees, etc. This indicates the need for “enhanced traders” (for lack of a better term) which maintain a very broad range of information on services and provide very flexible access to it. This paper describes plans for an “enhanced trader” prototype, which will support a complex, extensible data model and query language access by clients. Issues discussed include: (1) reasons for selecting a relational DBMS over an object DBMS for the trader repository, (2) techniques used to support a complex data model with dynamic extensibility, and (3) techniques used to support remote query language access in an object based open distributed system.

This paper has been submitted to the 1994 International Conference on Applications of Databases, to be held in Vadstena, Sweden, June 21-23, 1994.

APM.1162.01 16 November 1994

External Paper

Distribution:
Supersedes:
Superseded by:

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Data Management for an Enhanced Trader

Data Management for an Enhanced Trader

Gomer Thomas, Mike Beasley, Yigal Hoffner

APM.1162.01

16 November 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	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Data Management for an Enhanced Trader

Gomer Thomas†
Mike Beasley†
Yigal Hoffner
Architecture Projects Management (APM), Ltd.
Poseidon House, Castle Park
Cambridge CB3 0RD, United Kingdom
phone: +44 223 323010; fax: +44 223 359779
email: (cgt, mdrb, yh)@ansa.co.uk

Abstract

This paper describes the implementation of a “trader” in a distributed computing environment. The purpose of a trader is to provide client applications with information about services in order to enable interoperation. Requirements for the trader include the need to support a data model with a complex type hierarchy , the need for the data model to be dynamically extensible, and the need to support remote query language access. Among the important issues discussed are: (1) reasons for selecting a relational DBMS over an object DBMS for the trader repository, in spite of the advantages of an object DBMS for handling a complex type hierarchy, (2) techniques used to address the complex type hierarchy and dynamic extensibility requirements, and (3) techniques used to support remote query language access in an object-oriented distributed system. The overall, high level design of the trader is also described.

† Gomer Thomas is a Bellcore secondee to the ANSA project at APM.
† Mike Beasley is an ICL secondee to the ANSA project at APM.

1 Introduction

This paper describes work currently underway in the ANSA project[1] on developing a prototype for an enhanced trader, focusing on the data management aspects of the work.

A “trader” in a distributed computing network is an object which provides information to client objects about services which are available in the network,

1. The ANSA project (Advanced Networked Systems Architecture) is a collaborative industry effort to advance distributed computing. From 1985-88 it was funded under the UK Alvey programme, with the co-operation of 12 industrial partners. From 1988-93 it was funded under the EC ESPRIT programme, with 21 industrial partners. Since early 1993 it has been funded by a consortium of industrial sponsors. It influences industry directions in distributed computing through technology transfer to its sponsors and through input to key standards, such as the ISO Open Distributed Processing (ODP) Reference Model [ODP93] and the OMG CORBA specifications [OMG91].

for the purpose of enabling interaction between clients and servers. The original concept of a trader was that its primary function is to provide the information needed to support dynamic binding between clients and servers. The ANSA project is currently extending the concept of trading to encompass a wider range of information about services to support a wider range of activities required for successful interaction in a large scale, federated system. As part of this work, a prototype for an enhanced trader is being developed.

The repository for service information in the enhanced trader is based on a commercial relational database management system. Client objects get access to this service information through a query language (SQL) interface, as well as through certain pre-defined procedure calls.

Issues addressed in this paper include:

- the rationale for using a relational database management system,
- the information model and database design for the trader repository,
- the approach used to implement query language access in the context of a distributed system built around an object-oriented interaction model, and
- the overall design of the trader.

2 Trading

Trading has become a widely accepted concept in the distributed processing community, and is currently being standardized under the ISO ODP effort [ODP-Tr93]. This section describes the history of the trading concept and the enhancements currently envisioned by the ANSA project team.

2.1 The origins of trading

Trading has its origins in the concept of name services.

In general, a name service maps the name of an entity into a set of labelled properties of the entity. One especially important type of name service maps the name of a service in a distributed system into a network address where a server offering the service can be found. This name=>address type of name service is very important, because it allows dynamic binding of clients to servers. This means servers can be reconfigured dynamically without having to recompile the applications which access them. A good deal of research has been done on name services and their efficient implementation [CherMann84] [OppeDala83] [TePaRiZh84] [SchwZaNo87].

In the early days of the ANSA project it was recognized that dynamic binding could be enhanced considerably by a service which keyed on the full signature of the service, rather than the name [ANSA86]. This would allow a client to dynamically locate a server for binding, even if the client only knew the operations desired and not the name of the server offering them, and also check that the type of the interface offered by the server (allowable operations and their argument and result types) matched that expected by the client. Such a service was called a trading service, and a server for such a service was called a trader.

The next step was to extend the trading concept further by associating properties with services, and allowing clients to specify conditions on these property values to be used as a filter by the name server or trader when looking for matching services [Peterson87] [APM1005]. Examples of properties might be the maximum length of messages accepted for an electronic mail service, or the types of document formats supported for a document format conversion service, or the guaranteed response time. The name server or trader could also return values of these properties to clients to allow them to do their own filtering. It is in this form that the concept of trading has been incorporated into the ISO ODP Reference Model [ODP93] [ODP-Tr93]. In practice, the attributes usually have been viewed as <property-name, property-value>

pairs, where each property-value is a (relatively short) character string. (If a property has some other type, say integer, it would typically be encoded into a character string.)

2.2 The current trading model

There are two steps required for a client to get access to a server through a trader. First, the server “exports” a service offer to the trader. The service offer consists of:

- service access information (how to reach the server)
- service type information (signature of the service)
- service properties
- export policy (which clients can be told about the offer)

The service properties may include performance guarantees, availability guarantees, security guarantees, etc.

Second, the client “imports” an offer or set of offers from the trader. The client specifies the service type information for the desired service and perhaps some combination of properties. The trader returns to the client the service access information, and perhaps the service properties, for the server(s) matching the specification.

2.3 An extended trading model

Distributed computing networks are becoming steadily larger and more heterogeneous. The heterogeneity extends not only over technical aspects of the networks, but also over organizational aspects. It is increasingly common for a distributed computing network to be a federation of a large number of semi-autonomous sub-networks, each under the control and management of a different organization. The organizations are willing to co-operate to achieve

certain common objectives, but they are not willing to relinquish autonomy to a central authority.

In such an environment, new types of activities have to be supported:

- heterogeneous interoperation:
  Clients and servers need to interoperate in spite of differences in protocols, transaction models, naming conventions, exception handling, etc. The system should provide support for mechanisms to bridge these differences, perhaps including automated instantiation of gateways of various kinds.

- federated enterprise negotiation:
  Organizations often need to negotiate over the terms and conditions of interoperation, including access rights and restrictions, accounting/billing/payment arrangements, recourse if service guarantees are not met, etc. The system should provide on-line support for such negotiation, available 24 hours a day.

- federated development:
  Interoperating clients and servers are designed and developed by different organizations at different times. It is often difficult at best for the personnel in the different organizations to have very much person-to-person communication. The system should support the necessary transfers of information so these separately developed systems can in fact interoperate successfully.

- federated system management:
  It is necessary to have end-to-end performance management and trouble shooting. The various monitoring, failure reporting, and resource management mechanisms in the different parts of the system should be able to interoperate in order to provide it.

In order to support these activities, a trader needs to maintain a great deal of information about services, and clients must be able to retrieve this information in very flexible ways. The information to be maintained must include enterprise information, such as access rights and restrictions, accounting/billing/payment arrangements, contractual guarantees, etc. It must include detailed descriptions of the semantics of services, as well as signatures, so that designers and developers can accurately determine whether a service meets their needs and how to use it. It must include information on the protocols which can be used to reach the servers. Clients must have great flexibility not only for specifying what service(s) are of interest, but also for specifying what information they want about those services.

One way to view this situation is that traders must be able to support what might be called the “shopping” model, as well as the usual “matching” model. In the matching model a client gives the trader a complete specification of the service desired, including the service type description, and the trader returns to the client the service access information for one or more matching services, together with perhaps some additional predefined information on properties of the services to assist the client in selecting one. This model is primarily intended to support dynamic binding. In the shopping model a client gives the trader an incomplete specification of the service desired. The specification may or may not include the full service type description. The client also tells the trader what kinds of information is needed about the qualifying services. The client then takes this information and makes a decision on what service to use, based on some algorithm known only to the client. The client may not be interested initially in obtaining service access information. That may be requested later after a service selection is made. In fact, the client may first have to adapt to the service type of the selected service, so there may be considerable time lag between the initial query to the trader and the actual need to invoke the service.

3 Overview of prototype trader

3.1 Objectives for the prototype

Trading is a process, the process of helping clients and servers get in a position to interoperate. This process may be carried out by different configurations of components in different situations:

- Clients and servers may be able to interact directly with no need for a third party trading service at all.
- There may be a single monolithic trader which contains all the necessary information on all the services in the network.
- There may be a federation of traders, each of which contains all the necessary information on all the services in its sub-network.
- There may be a federation of traders, each of which contains all the necessary information on certain types of services in the network.
- There may be a federation of traders, each of which contains certain categories of information (enterprise information or semantic information or protocol information, etc.) on all the services in the network. (This option has a number of advantages in terms of implementing and managing the individual traders, but can create major consistency problems between the traders.)
- There may be various combinations of the above.

The goal of the ANSA project is to develop an architecture for open distributed processing. An important aspect of this is identifying functions which can be implemented once , and then re-used across many application domains. Trading is an example of such a function.

The objectives for developing an enhanced trader prototype are:

1. demonstrate the feasibility of developing a generic trader component which can be configured in different ways to meet trading requirements in a wide range of different situations,
2. explore how to configure such traders for different situations, and
3. explore how collections of such traders can co-operate together in federated systems.

3.2 Trading for information services

An interesting question is how trading should be handled for a service which is an information service or database, for example a bibliographic search service or a restaurant rating service. In order for a client to access an information service successfully, it is necessary for the client to know the logical data model for the information provided by the information 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 information 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 information server itself, or perhaps from a specialized data dictionary (which of course can be viewed as a specialized trader).

The current plan is that the initial implementation of our prototype trader will not contain full meta-information on information services. It will only contain some broad descriptive properties. The functionality of the prototype trader

may be expanded in this respect later, either by expanding its data model, or by linking it with an off-the-shelf specialized data dictionary of some sort.

3.3 Data management requirements for the prototype

The extended model of trading leads to three key data management requirements for the prototype trader's repository of service information:

- The information model must be extended.
  As noted above, it must accommodate many more categories of information than can be conveniently represented with a simple <property-name, property-value> model, since service information may be of numerous, diverse, complex data types.

- The information model must be extensible.
  Different types of services require different types of information to describe them. In a federated system new types of services constantly appear. It should be possible to accommodate new types of services without redesigning and recompiling the trader, or even taking the trader out of service to update the information model.

- Clients must have query language access to the trader information.
  The kind of flexibility which clients need for access to information in a trader can only be achieved through a query language, not through a set of pre-defined functions. The current draft of the ODP Trader standard envisions the specification of services in terms of boolean expressions in the properties of the services. This is of course a form of query language. One can achieve more generality, and probably easier implementation, by simply using a standard database query language.

4 Implementation approaches

This section discusses several of the key implementation approaches to the enhanced trader prototype.

4.1 Choice of distributed computing platform

One key decision in designing the prototype trader was what to use for the underlying distributed computing platform. The environment to be used is a CORBA-compliant ORB product [OMG91]. The primary factors in this choice were:

- the functionality which this provided.
- a desire to demonstrate the prototype in an environment which adheres closely to the ODP Reference Model.
- a desire to make it easy for the ANSA sponsors to port the prototype to other platforms, e.g., any other CORBA-compliant ORB product.

At the same time, a conscious effort will be made to modularize the design in such a way as to minimize the dependencies on the underlying platform.

4.2 Choice of data management system

Another key decision in designing the prototype trader was what type of data management technology to use. There were two obvious choices:

- relational DBMS and SQL for remote client access
- object DBMS and some object query language for remote client access

This was not an easy decision, since there are major arguments for and against each possible choice. It was eventually decided to use a relational DBMS and SQL. The primary factors in this selection were:

- Relational DBMSs have built-in support for dynamic SQL, so it will not be necessary to invent a query language or develop any software to get the DBMS to execute a dynamic query language operation. This is not true for many object DBMSs.

- The programming interfaces to relational DBMSs are reasonably standardized, so ANSA sponsors who want to use the prototype software will not be locked in to the DBMS vendor chosen for the prototype. This is generally not true for object DBMSs.

- A prototype based on a relational DBMS will demonstrate that it does not require exotic database technology to implement the enhanced trading model. It can be done with mature, commercial products.

- The ANSA project has ready access to a relational DBMS at very low cost.

An important disadvantage of this selection is that the complex types of data which need to be stored in the trader repository will be more difficult to manage with the relational data model than they would be with an object model. This problem is discussed at some length in section 4.3.

Another disadvantage is that SQL will be less convenient to use than an object query language, such as the language OQL developed by the Object Database Management Group [ODMG93], would be.

Because of this, a conscious effort will be made to design the prototype trader in such a way that it would not be too difficult to swap out the relational DBMS and swap in an object DBMS at some later time. There are two scenarios which might make such a swap feasible:

- support for an SQL interface by some suitable object DBMS

- support for a standard object query language such as ODMG's OQL by some suitable object DBMS, and conversion to use of that query language as the query language for clients to access the trader.

Other scenarios are not promising, because of the effort involved to develop an SQL to OQL translator.

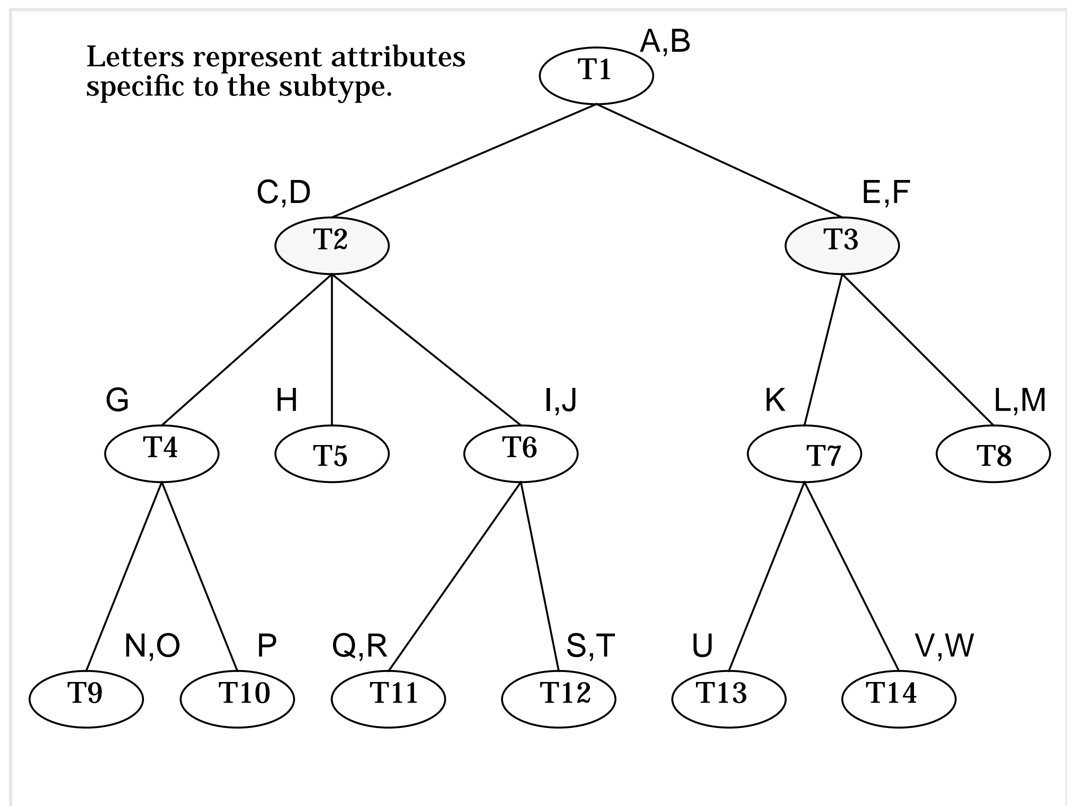
4.3 Database schema

The first step toward the design of the database schema for the enhanced trader prototype is to develop a logical data model of the information to be kept in the trader repository for generic services. That is, identify the types of information which are likely to apply across a wide range of services. Next, extend this logical data model to encompass information which applies to many commonly encountered specific classes of services.

The resulting logical data model can then be used as a basis for a relational database design in the usual way. However, a problem which arises is that the supertype-subtype hierarchy of the logical data model is very rich, which always creates a problem for relational database design. If one creates a table to represent every subtype in the hierarchy, one ends up with an impractically large number of tables. If one creates tables to represent only a few types chosen up near the top of the hierarchy, then each such table must represent the type chosen and all of its many subtypes. This means that each table will have a great many attributes, but for any particular entity which appears as a row in the table, most of the attributes do not apply.

For example, in Figure 4.1 one might create tables corresponding to just the shaded types on the second level from the top (T2 and T3). Then the table for T2 would need attributes (A,B,C,D,G,H,I,J,N,O,P,Q,R,S,T), and that for T3 would need (A,B,E,F,K,L,M,U,V,W). However, the entities in the subtype T5 are represented by the table for T2, but the only attributes which are meaningful for T5 are (A,B,C,D,H). Similarly for other subtypes in the figure.

Figure 4.1: Type Hierarchy



One way to get around this is to use a tag-value (<property-name, property-value>) representation for attributes which are specific to subtypes near the bottom of the hierarchy.

For example, suppose one has a table T with key K and other fixed attributes A, B, ..., and suppose one needs to represent additional attributes for the entities represented by this table. To do this one can have another auxiliary table TP with key K and attributes Pname, Pvalue.

Figure 4.2: Auxiliary Table for Extended Attributes

K	A	B	...
k0	a0	b0	...
k1	a1	b1	
k2	a2	b2	

K	Pname	Pvalue
k0	'X'	x0
k0	'Y'	y0
k0	'Z'	z0
k2	'X'	x2

To indicate that entity with key value k0 has attribute X with value x0, one puts the entry (k0, 'X', x0) in table TP. (See Figure 4.2.) One can retrieve the

attribute name and the attribute value for the entity by taking the join of T with TP. Attributes represented in such an auxiliary table are sometimes called extended attributes.

There are several disadvantages with this tag-value approach:

- It takes a join just to retrieve an extended attribute value. This may not present too much of a performance problem if the DBMS allows joint clustering of tables, but few DBMSs do.

- Querying these extended attributes requires a fundamentally different and more complicated SQL statement than querying ordinary attributes. One can allow applications to generate SQL queries which treat extended attributes just like ordinary attributes, and then translate them into valid queries against the actual schema before giving them to the DBMS to execute, but it is not straightforward to do so.

- If different extended attributes have different data types, then multiple auxiliary tables may be needed. Alternatively, one can use a single auxiliary table with key K and three attributes Pname, Ptype, Pvalue, where Pname is a character string containing the name of the attribute, Ptype is some kind of type designator for the attribute, and Pvalue is of type BLOB (binary large object) and contains the value. However, this alternative approach may make retrievals difficult, since the DBMS may not know how to do comparisons with BLOBs.

- The software which receives a service offer from a server and loads it into the repository must be rather complex, since it must distinguish those attributes built into the schema from the extended attributes and treat them differently.

Nonetheless, there may not be any other workable alternative.

Another problem is how to provide the required extensibility to the information model. There are essentially two kinds of extensions which are needed:

1. Create new entities or relationships in the logical data model.

2. Create new attributes to existing entities or relationships.

For the first kind of extension it is relatively straightforward to add new tables to the database. Normally this can be done without disrupting operations. For the second kind of extension one needs to add additional attributes to existing tables. Unfortunately, normally this cannot be done without disrupting operations. However, the tag-value approach described above can also be used to address this problem.

5 Remote query language access

Another interesting issue is how to handle the remote SQL query interactions between clients and the trader.

The computational model for the ODP Reference Model is that clients invoke operations on servers. Each operation has a signature consisting of:

- operation name
- number and types of arguments (input parameters)
- finite set of possible “terminations”
- number and types of results (output parameters) for each termination

At the engineering level, implementations usually require an Interface Definition Language (IDL) description for each operation, and both client and server are compiled against this definition. This type checking at compile

time, coupled with use of a trader at run time to assure that the client actually binds to a server of the type it is expecting, guards against type mismatches.

However, such an implementation does not mesh well with dynamically generated SQL queries (or dynamic queries in any other query language of comparable power). The number and types of the arguments and results of an SQL query are not known in advance for dynamic queries. They are determined by the text of the query itself, which is often generated at run time.

The same problem arises in many distributed computing environments based on RPCs, such as OSF DCE [OSF92], even when they do not have the object oriented character of the ODP Reference Model.

There are several approaches one can take to dealing with this problem:

At one extreme one can define a remote operation “query” with a single input parameter, which is a byte string of indeterminate length, and a single output parameter, which is another byte string of indeterminate length. This operation can be used as the vehicle for all query language invocations. Some suitable remote database access (RDA) protocol, such as the ISO RDA protocol [ISO-RDA1] [ISO-RDA2], can be encoded inside these byte strings. This amounts to treating the remote operation protocol as simply a transport protocol for the RDA protocol.

At the other extreme one can adapt the remote operation infrastructure so that it can handle dynamically defined operations. A query language statement is treated as an operation constructor. In the context of a particular database schema it defines the signature and semantics of an operation. For example, the operation defined by a SELECT statement is the creation of a new virtual table. The operation defined by a DELETE statement is the deletion of the specified row(s). If the query statement is presented to the

DBMS, the DBMS can determine the type of the operation and return that information to the client. (This is essentially what the SQL "DESCRIBE" statement does.)

As an intermediate approach, one can define a set of remote operations corresponding to the X/Open CLI specification [CLI92] or the Microsoft ODBC specification[ODBC92] or something similar. Each CLI call would then map to the corresponding remote operation. Since some of the CLI calls have a dynamically defined number of input and/or output parameters, this would still require in some cases encoding of the actual parameters inside byte strings of indeterminate length.

For the enhanced trader prototype the dynamically defined operations approach is the one being used. A data retrieval query is viewed as a two stage process. In the first stage the query is logically executed, forming the result at the server. In the second stage the results are retrieved from the server. (The first stage corresponds to an SQL "OPEN CURSOR" statement. The second stage corresponds to an SQL "FETCH" statement.)

Given an SQL retrieval query, the client first invokes a "describe" operation at the server. The input to the "describe" operation is a character string, the text of the query. The output from the "describe" operation is an operation-type (an object of *type* "type"), telling the client the type of the operation defined by the query statement. (This step can of course be skipped if the client already has enough information about the query to figure out the operation-type for itself.)

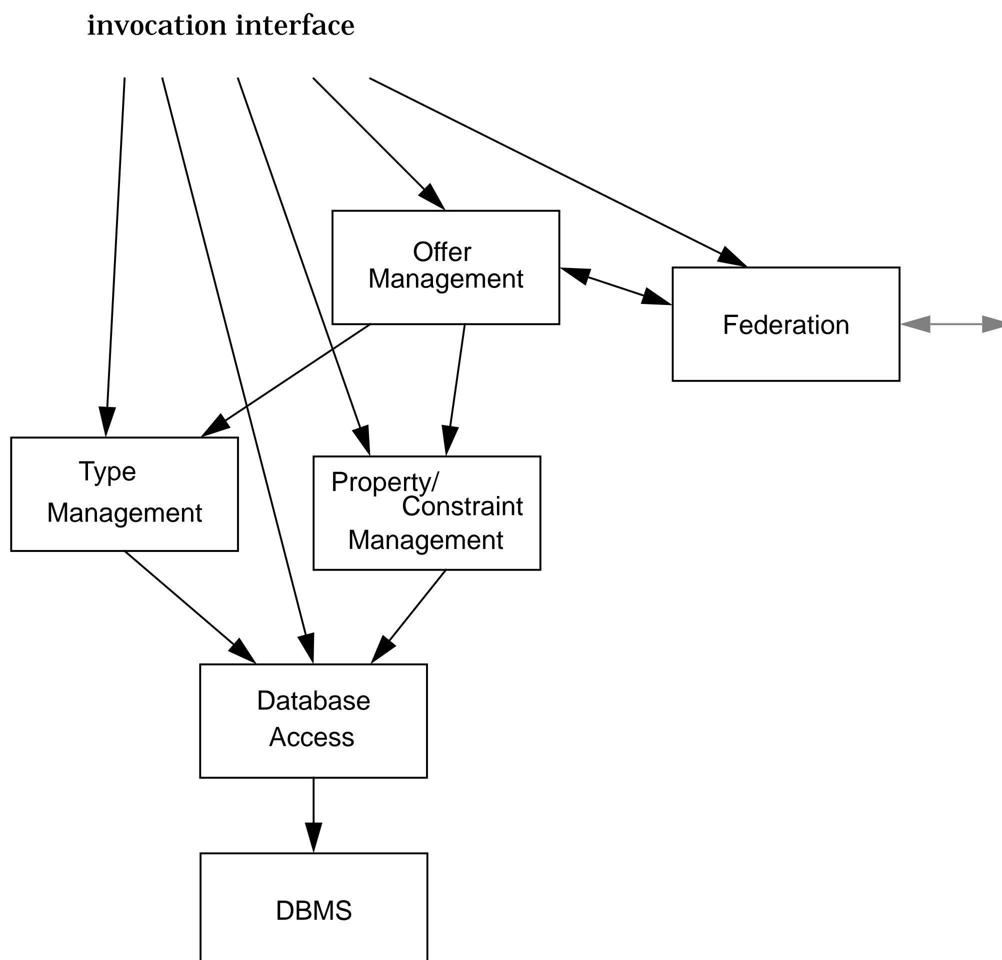
The client can then invoke the actual query operation, passing in the appropriate input parameters. The result of the query will be a set of objects of some type determined by the query statement, and the client will be passed back a reference to that set (which is itself an object). The client can then invoke "fetch" or "iterate" operations on that set to retrieve the individual result objects.

Another paper is in preparation which describes in considerably more detail this model for invocation of query language operations in an object oriented distributed system [Thomas94]. The overall model of invocation is the same for essentially any query language, SQL or OQL or whatever.

6 Trader design

Figure 6.1 shows the modules of the trader.

Figure 6.1: Modules of the Trader



The invocation interface is where invocations of trader operations first arrive at the trader. These fall into three general categories:

- requests by servers to register or delete or modify service offers

- requests by prospective clients to get information about service offers
- requests by the trader administrator(s) to manage certain aspects of the trader configuration, particularly links to other co-operating traders

The Offer Management module implements operations to register new service offers with the trader and delete service offers, and possibly some pre-defined operations to look up service offers by commonly used criteria (e.g., signature).

The Type Management module is computationally separate from the other modules, but it will be co-located with the other modules from an engineering point of view. It supports the operations for adding type definitions, deleting type definitions, listing type definitions, and comparing types for compatibility.

The Property/Constraint Management module supports definition of new properties and their allowable values, and comparison of property values.

The Federation module handles cooperation with other traders, including operations to be used by the trader administrator to set up and shut down links to other traders.

The Database Access module handles access to the repository of service offers. One of its main functions from a design standpoint is to provide a portability layer between the rest of the trader software and the DBMS.

7 Summary

This paper has described work in progress on an enhanced trader prototype. The current status is that most of the sticky technical problems have been analyzed, as indicated in this paper. A very high level view of the logical data model has been developed, and refinement is under way. A high level view of

the design has been developed, and refinement is under way. The plan is to have an operational prototype by June of 1994.

8 Acknowledgements

The authors would like to thank the many members of the APM team for many helpful discussions on the topic of this paper, especially Jane Cameron, Gray Girling, Andrew Herbert, David Iggulden, Rob van der Linden and Andrew Watson.

References

[ODP93]

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

[OMG91]

The Common Object Request Broker: Architecture and Specification (Release 1.1), Object Management Group, December 1991.

[ODP-Tr93]

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

[CherMann84]

D.R. Cheriton & T.P. Mann, Uniform access to distributed name interpretation in the V-system, Proc. 4th Int Conf on Distributed Computing Systems, pp 290-297, 1984.

[OppeDala83]

D.C. Oppen & Y.K. Dalal, The Clearinghouse: a decentralized agent for location named objects in a distributed environment, ACM Transactions on Office Information Systems, 1(3), July 1983.

[TePaRiZh84]

D.B. Terry, M. Painter, D. Riggle & S Zhou, The Berkeley Internet name domain server, Proc USENIX Assn Summer con, pp 23-31, June 1984.

[SchwZaNo87]

M. Schwartz, J. Zahorjan & D. Notkin, A name service for evolving heterogeneous systems, University of Washington Dept of Computer Science Tech Report 87-02-05, Feb 1987. (integrating name servers)

[ANSA86]

ANSA Functional Specification Manual (Release 1), May 1986.

[Peterson87]

L.L. Peterson, A yellow-pages service for a local-area network, Proc SIGCOMM'87 Workshop, Aug 1987, pp. 235-242. (attributes as well as names)

[APM1005]

The ANSA Model for Trading and Federation, Architecture Report, APM Ltd., Cambridge UK, APM.1005.01, February 1993.

[ODMG93]

R.G.G. Cattell (editor), The Object Database Standard: ODMG-93, Morgan Kaufmann, 1994.

[OSF92]

OSF DCE Application Development Guide, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142, USA, 1992.

[ISO-RDA1]

ISO 9579-1:1992, Information Technologies - Open Systems Interconnection - Remote Database Access - Part 1: Generic Model, Service and Protocol, November 1992.

[ISO-RDA2]

ISO 9579-1:1992, Information Technologies - Open Systems Interconnection - Remote Database Access - Part 1: Generic Model, Service and Protocol, November 1992.

[CLI92]

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

[ODBC92]

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

[Thomas94]

G. Thomas, Remote database queries in object oriented distributed systems, paper in preparation.