Interworking of Traders in a Distributed Computing Environment *

Michael J. Katchabaw†, Meeta M. Khurana†, James W. Hong‡ and Michael A. Bauer†
† Department of Computer Science,
University of Western Ontario, London, Ontario, Canada
‡ Department of Computer Science and Engineering,
Pohang University of Science and Technology, Pohang City, Korea

Abstract

Distributed computing systems are composed of various types of services. Providing a reliable and efficient distributed computing environment largely depends on the effective management of these services. One important aspect of managing services within a distributed computing environment is maintaining information about the services, especially at run-time.

To facilitate this, much work has been done on mechanisms called *Traders* that provide a framework for managing and exchanging services in an open distributed computing environment. Open Distributed Processing (ODP), a proposed ISO standard, includes such a trader mechanism. This paper addresses problems associated with providing a single transparent trading service in the presence of multiple physical traders in a large distributed computing environment. The group of traders form a *federation* or *interworking* of traders which share their service offer spaces. We present the design for the interworking of traders, as well as a prototype implementation using OSF DCE, and report our experience with it to date.

1 Introduction

Many of today's computing environments are composed of geographically dispersed computing systems interconnected through communications networks. Such computing environments will require a variety of distributed services to perform various tasks on behalf of users and applications. As the size and heterogeneity of these computing environments increase, so too will the number and type of services. Since system users depend on these services, their reliability and availability will be critical. This requires an effective management of information on those services, such as keeping track of what services are available, where they are located, what their properties are, and what their statuses are. This is especially important in a distributed environment where systems come and go, servers are migrated or replicated, etc.

The need to maintain information about the services in a distributed computing environment has been recognized by others; much of the work on naming systems [5, 6, 7, 8, 17] has been motivated by this need. Others have also recognized the need for information about services available during runtime [1, 16]. In particular, the ISO has begun work on a proposed standard for Open Distributed Processing (ODP) [11]. Included in this proposed standard is a mechanism called the *Trader*, which provides a framework for "trading" services in an open distributed computing environment [12]. "Trading" is a term that is defined as the sharing of services between entities (or objects) in a distributed computing environment. Although there has been some work on the refinement of the Trader [3, 9, 15] and investigation of its potential uses [18, 20, 21], more work is required for it to become an acceptable international standard.

The ODP Trader draft standard also specifies a high-level view of *interworking* (or *federating*)

*This research work is supported by the IBM Center for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada. The IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, IBM Canada Ltd., Dept. 21/894, 844 Don Mills Road, North York, Ontario M3C 1V7.

Appeared in the Proceedings of the 1995 IBM Centre for Advanced Studies Conference, Toronto, Canada, November, 1995

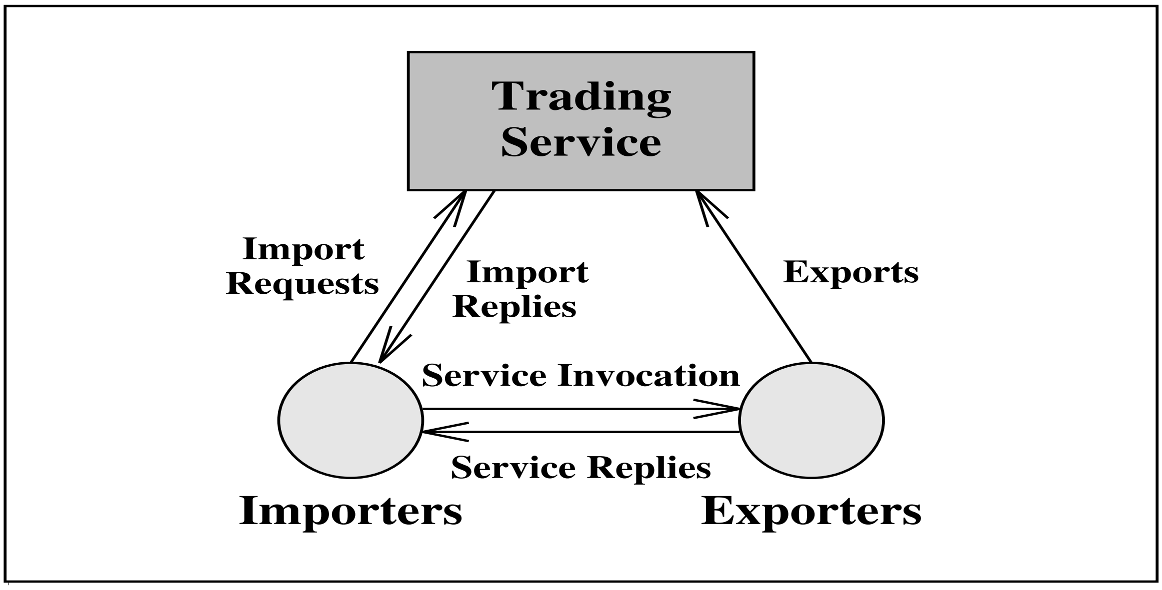

Figure 1: ODP Trading Service and its Clients

two or more traders for the purpose of making the trading service transparently and widely available across domains in a large distributed computing environment [12]. By interworking traders, the benefits of traders are extended further, allowing access to a wider variety of services and improving resource sharing between domains. Some have started looking at the issues involved in interworking [23]. Other earlier attempts at the interworking of traders have been made using X.500 [2, 19]. Our work attempts to interwork traders based on ODP concepts using direct trader-to-trader communication. We achieve this by extending the trader interface to accommodate interworking. Our main interest in the interworking of traders is to gain an understanding of the complexities involved in the interworking through design and prototype implementation of interworking traders based on the current draft standard. We wish to communicate our experience to the developers and users of the ODP framework and others involved in research into open distributed systems.

In this paper, we address some of the key issues and challenges in providing a transparent trading service in a large distributed computing environment. In particular, we discuss the design of interworking traders, which will present a single homogeneous trading service at the logical level but which will involve multiple physical traders cooperating with each other at the implementation level. We present an architecture of the interworking traders and its major components. We also describe a prototype implemented on top of OSF DCE [16] and report on our experiences with it. Although our prototype implementation does not fully implement all the features recommended in the standard draft, our approach is proving to be very simple and efficient, as well as a solid basis for further investigations.

The rest of the paper is organized as follows. In Section 2, we present an overview of the ODP Trader and interworking. Section 3 discusses general requirements for the interworking of traders in a distributed computing environment. Section 4 presents our design for the interworking of traders. Section 5 describes our prototype implementation effort using DCE. Section 6 provides an example of the use of our interworked traders using a client-server application. Section 7 concludes this paper by summarizing our work and discussing possible future work.

2 Overview of the ODP Trader and Interworking

The ODP framework is a set of draft standards [10, 11, 13, 14] that are aimed at a variety of architectures, networks, and operating systems to provide an open distributed processing environment. The ODP Trader is one component of the ODP environment, whose purpose is to provide a match-making facility between entities (or objects) providing services and those needing the services in ODP [12]. The ODP Trader draft standard discusses a large number of components that comprises a trader. Some of these deal specifically with trading policies, security requirements, accounting requirements, transfer requirements, quality of service, and interworking.

At the core of the ODP Trader system are the interactions among four different types of objects: services, traders, exporters, and importers (see Figure 1). A *service* is a function provided by some ODP object for use by other ODP objects. To make services available between other objects, a *trader* is used to facilitate the trading or sharing of services. An *exporter* is an ODP term for a service provider. It is an object with a service that it wishes to make available to other objects. The exporter may be either providing the service itself or acting as an agent on behalf of the service provider. Providing a service is accomplished by *exporting* the service to a trader. An exporter is also able to later *withdraw* (make unavailable) the service. In ODP terminology, a requester of services is known as an *importer*. The expectation is that importers in the ODP environment can operate without any prior knowledge of where the required services are or which object provides them. To find these services, the importer must make a service request to a trader. The trader then returns to the importer the details of the services matching the service request if any exist. Exporters and importers are collectively referred to as *clients* of the trading service.

Services are exported and imported through the use of *service offers* and *service requests*. A service is exported in the form of a *service offer* which describes the service being made available. This description includes a set of attributes that the service possesses, as well as a specification of the interface to the service. An importer discovers services by sending import requests to traders. The main component of an import request is the *service request*, which is a set of assertions that describes the desired service. The import request also provides information describing the method and scope of the search to be used by the trader.

The Trader draft standard also defines *interworkings* of traders, which are linked to share their service offer spaces without loss of individual autonomy [12]. This function is necessary in a very large distributed computing environment which may comprise a number of different organizations with a heterogeneous hardware and software platforms. Interworking of traders allows the users to import services transparently from their own trader as well as from other traders.

The group of interworking traders is linked to form a directed trading graph, which forms a knowledge graph of the next trader to be accessed in a traversal path for an import operation. Figure 2 illustrates an example of a trading graph of five interworking traders. Each trader in a group of interworking traders, however, has its own global search policy (such as the extent of the propagation of a search request, sequential versus concurrent searches of remote trader) and domain boundary crossing policy (such as type domains, security domains and technology domains).

3 Requirements for Interworking Traders

In our investigation into the interworking of traders, we have identified four primary requirements on which to base our system: a functional architecture for the traders, a protocol that allows sufficient client-trader and trader-trader communication, a simple set of service interfaces, and repositories for storing service, client, and trader information.

3.1 Functional Architecture

Certain functionality must be provided by the traders to satisfactorily handle both the requests of clients and other traders. The traders must have components to communicate with clients and other traders, parse the incoming requests, and interact with stored service, client, and trader information to carry out the requests.

Since this is an early attempt at interworking traders, the architecture should support future modifications and additions. As a result, the architecture must have a certain degree of flexibility and expandability and provide a good foundation for building upon at a later date.

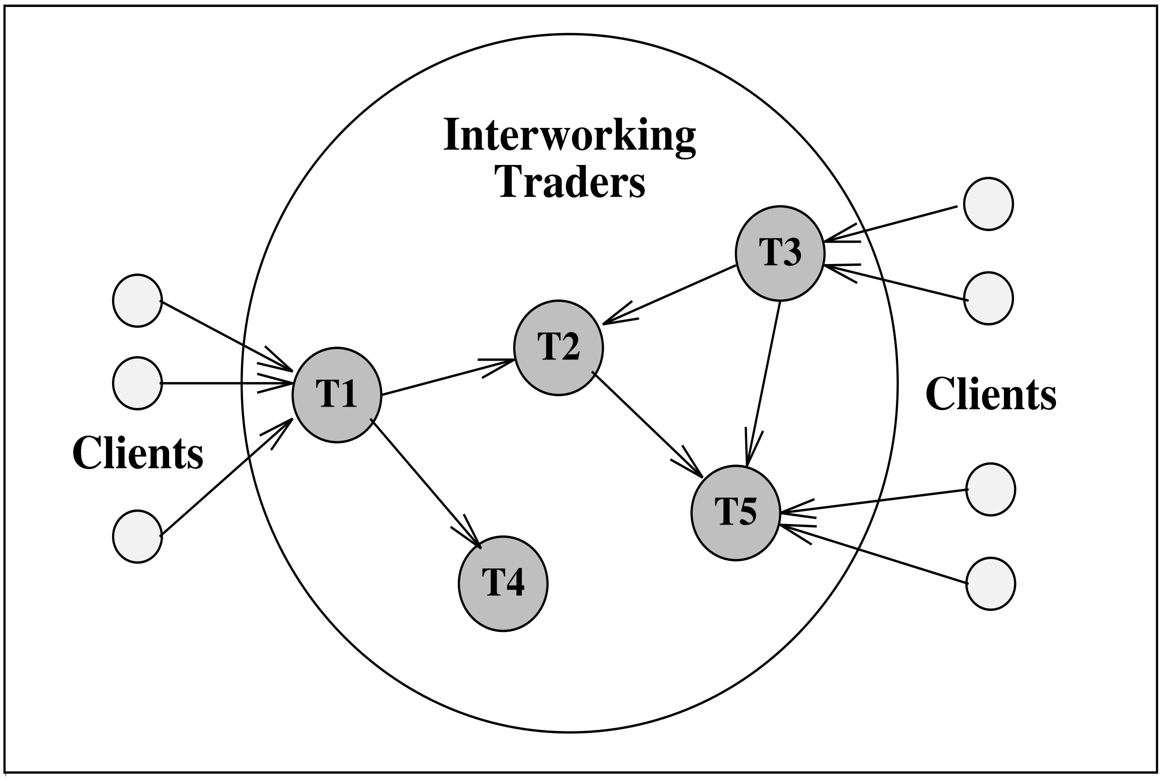

Figure 2: An Example Trading Graph of Interworking Traders

3.2 Communications Protocol

Clients need to communicate with traders. To work effectively, a well-defined communications protocol must be established between client and server which provides a set of message types recognised by both ends of communication. Furthermore, a protocol must be defined to allow traders to communicate with one another, to implement the interworking.

The communication protocols should be simple, yet powerful enough to support the messages required to allow clients to access service information both locally (through the local trader) and globally (through the network of traders).

3.3 Service Interfaces

The trader service interfaces should provide simple access to a trader for clients and other traders. The following types of interactions will be involved between clients and traders and thus interfaces should be provided to support them.

General Client Interactions: All clients will need a method of establishing a link with a local trader and later breaking that link when the service is no longer required.

Exporter Interactions: A trader must provide interfaces through which an exporter may advertise its services by adding, modifying, or deleting a description of its services it wishes to make available.

Importer Interactions: A trader must provide a method which allows an importer to access information on various services advertised by exporters.

Trader Interactions: A trader must provide an interface to which other traders can make requests, on behalf of their clients.

3.4 Service, Client, and Trader Information Repositories

The repositories required for the system must contain information pertaining to the services that are being exported and imported using the traders, the clients that are requesting those services, and the traders participating in the interworking. To do this, the repositories must provide the capability to add, modify, and delete the information that they store. The emphasis of this work is not to provide a complex data modelling, but rather to evaluate a mechanism for the interworking of traders. Consequently, a simple approach is provided by the system proposed. In larger scale, more full-featured implementations, more complex modelling can be obtained by simply enhancing the repositories.

4 Design of Interworking Traders

The previous section presented the fundamental requirements for the interworking of traders. This section outlines the general design for an interworking system that satisfies these requirements in providing trading and management of resources to clients in a distributed computing environment.

4.1 System Architecture

The trading services should be completely transparent to the end user. It is irrelevant to the user where services for their applications are located (either local or remote). Moreover, the user is not concerned with how the requested services are accessed.

Similarly, the programmers of these applications making use of the trading services need not worry about its underlying architecture, merely that the requested functionality is provided. To them, they would see only that some exporter is providing a service that their application needs. To locate it, their application would contact the local trader, import the interface to this service, and then contact it directly. They would be completely unaware that the interworking existed or that their request was fulfilled by some other trader, if the local one could not satisfy it. Hence, the top-level design of the trading service should hide the details of the architecture of the application and provide this simplified view. As a result, the top-level view for this application would appear as in Figure 1. In actuality, the trading services shown in the above architecture will be provided by a network of traders. To the user of the system, this would still appear as a single entity. This is shown in Figure 2.

With these top levels specified, the lower level design we produce must support this abstraction. At this level, we must also separate the application into logical modules as well, to make further design and later implementation easier and less problematic. The lower-level design architecture presented in Figure 3 meets those goals. We now proceed to discuss the design of each component from this architecture in more detail.

4.1.1 Client Interface

The client interface is the component of the system that makes requests for services from a trader. Clients can either be exporters advertising services, importers requesting services, or both. This module contains all of the essentials required for communication with traders - the marshalling of requests, the sending of requests, the receiving of responses, and the unmarshalling of the replies.

4.1.2 Trader Interface

The trader interface waits for incoming requests from client processes, accepts them and processes them by translating and forwarding the requests on to the service information repository. The repository would then return a result to this module, which would then translate this result and send this back to the appropriate client as a reply. If the repository could not satisfy the request, this module will forward the request on to neighbouring traders and wait for a response. Information regarding neighbouring traders can be retrieved from the trader link repository. The trader, in forwarding the request, takes on the role of a client to the neighbouring traders. When done, the result of the client's request would then be returned to the client process. Naturally, the trader would also have to authenticate the client's request using information stored in the client information repository.

4.1.3 Service Information Repository

The service information repository is the module that contains information on the services that have been exported to a trader by client applications. This component provides a set of basic operations that allow the trader to access and manipulate the service information and then forward replies to the appropriate clients. To extend this system to general re-

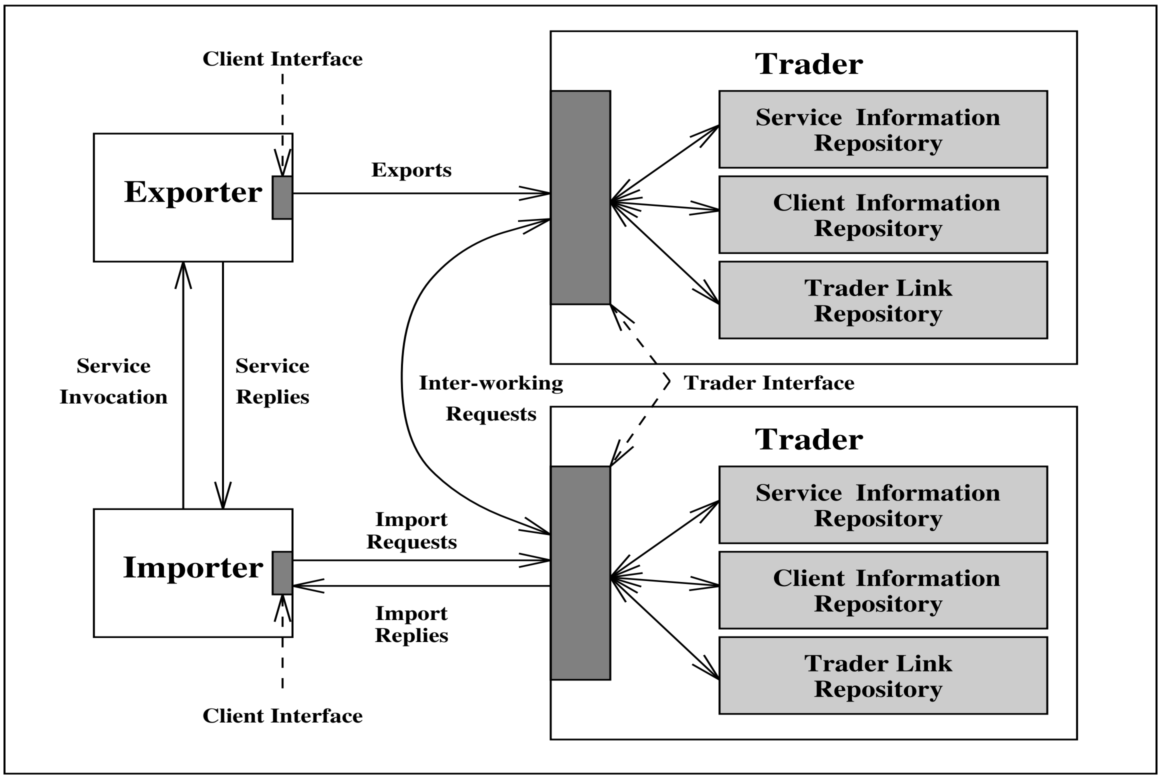

Figure 3: Low-Level Architecture for Interworking Traders

sources with complex modelling, only this module need be replaced.

The repository can basically be divided into two sub-modules. One sub-module would be the data structure that stores the information and its access mechanism. This could be internal or external storage. Regardless, this data structure would contain, for each service, the service's name, a unique handle for the service, the handle of the client that exported the service, the service's type, the service's attributes, and the interface to the service. This interface may be a presentation address (host and port number) or the necessary information to contact the service using a remote procedure call (ONC, DCE, etc.). The second sub-module would be a service interface that provides a set of operations to access the data structure containing the information through the structure-specific access mechanism. This would be composed of routines for adding, deleting, modifying, or retrieving service information in the structure to support the operations that the trader must provide. Matching inside the structure is attribute based rather than name based. That is, clients can search for services based on a collection of attributes that the requested services must have in order to be selected instead of only based on services' names.

4.1.4 Client Information Repository

The client information repository is the module that contains information on the clients that are making use of the trader, including both exporters and importers. This component provides a set of basic operations that allow the trader to access and manipulate the client information for the authentication of requests being made to the traders by the clients.

Like the service information repository, this component can be divided into two sub-modules, one containing the data structure that stores the information and one that provides the service interface that allows access to this data structure. The data structure will contain a set of records, one for each client, containing the client's name, originating host, and a unique handle for the client. The service interface will allow operations to add clients to the structure, remove clients from the structure, and validate a client's handle. These services will allow a very basic authentication mechanism.

4.1.5 Trader Link Repository

The trader link repository is the module that provides access to information on neighbouring traders in the interworking of traders. Once again, a set of basic operations are required to access and manipulate this information.

Like the previous two repositories, this component can be divided into two sub-modules, one containing the data structure storing the information and one that provides the service interface that allows access to this structure. The structure would store a set of records, one for each neighbouring trader, including the name of the trader, and the host at which the trader can be contacted. The structure also contains information on the trader itself, including a unique identifier for the trader within the federation. Information regulating access to services through interworking could also be stored in this repository (for example, restricting the sharing of services with certain traders, etc.). The interface to this structure would allow the addition of neighbours, deletion of neighbours, and fetching information on neighbours (both listing and specific retrieval). This information provides the required information to allow interworking requests.

4.2 Protocols and Service Interfaces

Now that the architecture of the interworking trader system has been presented, we can discuss the protocols and service interfaces used for communication. These interfaces have been grouped by function, namely client-trader, importer-trader, exporter-trader, and trader-trader. At this point, there is still much on-going work on trader interfaces, particularly where interworking is concerned [22]. As a result, we decided to develop a simple interface (i.e. not completely ODP compliant) to allow experimenting with interworking traders with sufficient modularity to allow upgrading to a more standardized interface when one becomes accepted and available.

4.2.1 Client-Trader

Before any client can make use of a trader, the client must register itself with the trader. Similarly, when a client is finished using the services of a trader, it must deregister itself from the trader. While these operations are not required for the trading system, they are useful for keeping track of clients and providing a simple authentication mechanism. In the future, as the need arises, this information can be used to implement more sophisticated security measures.

register: The operation called *register* allows a client to identify itself to the trader to get a client handle. In a given session, a client may use both importing and exporting services; as a result, the client need not state what use it will make of the trader.

```
register([in] client_information,
         [out] client_handle);
```

deregister: The operation called *deregister* allows a client to deregister itself from the trader using its unique handle.

```
deregister([in] client_handle);
```

4.2.2 Importer-Trader

Importers are trader clients that make requests for information on services that the trader is advertising on behalf of exporter clients. As a result, the importer service interface should include all of the operations needed to access this information. (The `trader_depth` parameter required for each operation is discussed later in Section 4.2.4.)

search: The operation called *search* is used to obtain a list of services matching a set of attributes, up to a certain match limit. References to these services are returned by the trader.

```
search([in] client_handle,
       [in] service_attributes,
       [in] match_limit,
       [in] trader_depth,
       [out] service_list);
```

list: The operation called *list* is used to retrieve a list of attributes for a given service, based on a service handle obtained by the *search* operation. This allows the importer to decide which service best meets its needs.

```
list([in] client_handle,
     [in] service_handle,
     [in] trader_depth,
     [out] service_attributes);
```

select: The operation called *select* is used to retrieve the interface to a service, based on a service handle obtained by the *search* operation.

```
select([in] client_handle,
       [in] service_handle,
       [in] trader_depth,
       [out] service_interface);
```

4.2.3 Exporter-Trader

Exporters are trader clients that want to share access to services with other clients by advertising them using the trader. Although other clients have access to information on services exported, only the exporter has control over the service and may update or withdraw it at any time.

export: The operation called *export* is used by an exporter client to make a service available to other clients.

```
export([in] client_handle,
       [in] service_information,
       [out] service_handle);
```

withdraw: The operation called *withdraw* is used by an exporter client when the client no longer wants a previously exported service to be available to other clients. This does not imply that the service is shut-down, only that it is not available to importers. Only the exporter may withdraw its services (this is verified using the client handle parameter).

```
withdraw([in] client_handle,
         [in] service_handle);
```

update: The operation called *update* is used by an exporter client when the client needs to change the information stored by the trader on a service it has previously exported. This could be accomplished by a withdraw operation, followed by another export, but the update operation allows the service to retain its handle. Only the exporter may update its services (this is verified using the client handle parameter).

```
update([in] client_handle,
       [in] service_handle,
       [in] new_service_information);
```

4.2.4 Trader-Trader

Trader-trader communication is obviously required for the interworking of traders. However, it is not obvious as to how this communication should take place. One possibility would be to implement an entirely new protocol just for this communication. Another alternative is to use some kind of distributed information repository for storing service information and using its distribution properties and protocols to provide the interworking (this could be done using X.500 [5, 6]). Examining the above service interfaces, it is apparent that a trader could access another trader using the same interface. In effect, the trader would become a client to the other trader that it is making requests to, through direct communication. As a result, the above service interfaces suffice for trader-trader communication as well. Furthermore, only importer operations are really required for interworking, as clients from other domains should only be retrieving information from the local trader and not modifying it. As a result, we can simply make use of the proposed service interfaces for client communication between traders.

Links from trader to trader can be traversed in a very simplistic manner. When a trader cannot meet a request itself, it goes through each trader in its trader link repository, consulting them until the request is met. This is done by sending the appropriate client request to the neighbouring traders using the trader's identifier as the client handle. The neighbouring traders, receiving the request, respond to it as any other client request, and return results of the request to the originating trader. If these traders cannot meet the request, they forward it to their neighbouring traders in a similar manner. Contacting other traders can be done either sequentially or in parallel.

To prevent infinite looping and deadlocking of the system, the trader depth parameter in the interworking service interface limits the level to which client requests are forwarded. Because a finite bound is fixed on each request, each request can be serviced without these problems. However, this can become very complicated and time consuming in large interworkings of traders. An alternate approach would be to use some kind of "visited list" indicating the traders that have already seen this request or to have traders keep a buffered list of requests that have been processed recently to resolve these problems. Doing so would introduce new complications and problems at the same time. Because of the modularity and flexibility of the design developed, these methods can be easily examined and compared in terms of performance and efficiency with only minor changes to the prototype, which is discussed in the next section.

5 Prototype Implementation

A prototype for the interworking of traders was developed to determine the feasibility of our above discussed design. In this section, we discuss the details of the implementation of our prototype.

The prototype was developed in one of the department's laboratories, a network of 20 IBM RS/6000 machines running OSF Distributed Computing Environment (DCE) [16]. DCE allowed us to take advantage of its RPC mechanism for client-trader and trader-trader communication and its multi-threading capabilities to allow multiple requests to be processed concurrently. As mentioned in the previous section, both client-trader and trader-trader communications made use of the same procedure call interface, allowing interworking to take place seamlessly. The DCE RPC IDL definitions for our prototype implementation follow closely to the service interfaces given in Section 4.2. The definitions we are using are fairly simple; work on more thorough and complex definitions has also been done [22]. Through a modular design, we could adopt the use of these definitions when they are completed and accepted. Status checking and explicit binding in all procedure calls allowed for a certain degree of fault tolerance in the communication (for example, if one of the interworking traders went down and was restarted, its neighbours would be able to easily re-establish communication).

Client and service handles for interworking procedure calls were implemented as 32-bit integers. For both types of handles, the first 2 bytes specified the unique identifier for the trader in the interworking of traders (stored in the traders' trader link repositories). The last 2 bytes give the local client or service identifier at that trader. The resulting handle is guaranteed to be unique, providing that trader identifiers are uniquely assigned to all traders in the interworking of traders and that each client or service is assigned a unique local identifier. Because the prototype is intended towards investigation and experimentation, we can guarantee these conditions, giving unique identifiers. Also, given the limited size of our testing environment, we will have enough handles for our purposes. In more widespread use, this identification mechanism would naturally need to be enhanced to provide not only more handles, but better authentication as well. Both of these changes could be easily made in our existing prototype.

The information repositories were implemented, for simplicity, using flat files with a caching mechanism to enhance performance. This caching mechanism stored information from all requests, both those handled locally and those handled through requests to remote traders. This cache was consulted before each operation; only when the requested information was not in the cache was the flat file or neighbouring traders consulted. To ensure cache consistency, the caches were periodically refreshed by a separate thread running in each trader. To prevent loss of data due to failure, the caches followed the write-through technique, writing each data modification out to the database files as it was made. Each trader was responsible for managing and maintaining its own flat file repository. Alternative repositories, such as directories or databases, are under consideration for extensions in the future as they could be more powerful and flexible and would not impact the other components of the trader due to its modularity.

6 Experience

To illustrate the functionality of the interworking of traders, we demonstrate the use of the transparent trading mechanism using a simple client-server application. The application we make use of is a student information database system which was developed earlier. The client-server communication in this sample application was implemented using ONC RPC [4] and was augmented to include the DCE RPC client interface to allow interaction with the traders. To make use of the trading service, the student database server registers itself and exports its service, so that student database clients, not knowing where the server is located, can consult the trading service to determine its location.

To create a network of interworking traders, traders are started on each of five nodes in our AIX DCE laboratory (whose host names are `epitaf`, `euphoric`, `owzat`, `riffraff`, and

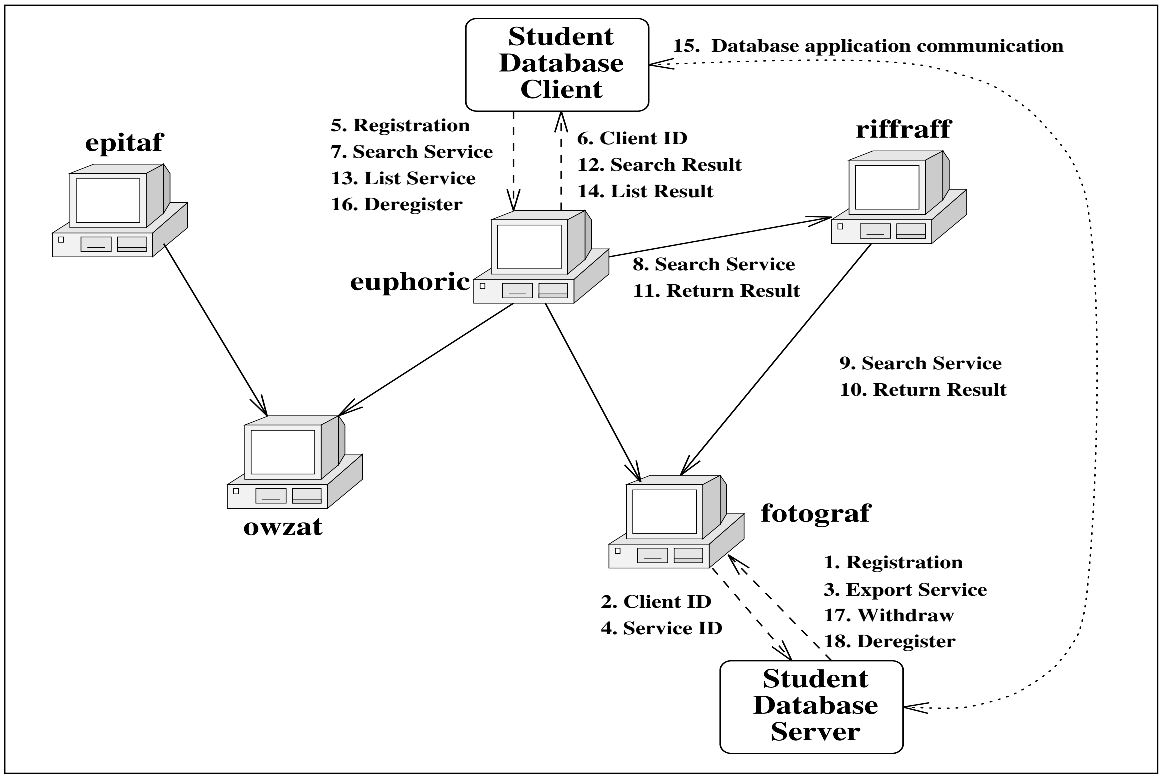

Figure 4: Student Database Client-Server Application Using Interworking Traders

`fotograf`, as shown in Figure 4). Solid arcs connecting the hosts represent links among the interworking traders, broken arcs represent client-trader interactions, and the dotted arc represents the student database application communication.

To demonstrate the interworking of traders, the database server and client are intentionally executed on different machines. Once they come up, they become clients of their local traders by first registering with them. The database server registers with its trader running on `fotograf` and exports its service, namely the "student database" service. The database client registers with its local trader running on `euphoric` and requests for a "student database" service with the desired service properties. Since the trader on `euphoric` cannot satisfy its client's request, it makes an interworking search request to one of its neighbouring traders, on `riffraff`, for example. The trader on `riffraff` cannot satisfy the request and thus makes a similar interworking search request to its neighbour on `fotograf`, which successfully finds the searched service information from its local database and returns the result to `riffraff`. The result is eventually returned to the originating trader and to its client. With the service information obtained from the trading service, the client goes ahead with its application communication with the database server. More detailed steps of these interactions are given in Figure 4.

Although not presented here, we have several other applications that have been either developed or modified to use the interworking trading service presented in the paper. As a result, we can conclude that our implementation does sufficiently implement the functionality of interworking traders. Performance analysis indicates that requests can take anywhere from less than a second to a few seconds, depending on the configuration of the traders (which traders are neighbours to other traders), the depth of the request being made, the location of the service being requested, and the result of the request (success versus failure). It was found that the caching mechanism used did indeed improve performance of the traders. When a sufficient collection of trader-enabled applications is available, a more thorough performance analysis can be performed to determine the effects of various factors on the system.

7 Concluding Remarks

While the study of traders in general is not new, the interworking of traders to provide a transparent trading service is an area still in the early stages of research and development and clearly warrants further investigation. In this work, several key ideas to the interworking of traders were introduced and discussed. The basic requirements of interworking were presented, focusing on a functional architecture, communication protocol, and service interfaces. Based on these requirements, an interworking trader system was developed. Both high-level and low-level designs were presented, together with client-trader and trader-trader protocols. It was discovered that trader-trader communication is quite similar to client-trader communication; hence, our design focused on making use of this observation.

This design was successfully implemented as a prototype system, and several sample applications were developed and tested using this prototype. Hence, we can conclude that interworking is possible using our design. While our current prototype is implemented using DCE, it can be easily ported,with minor changes, to other environments such as ANSAware, etc.

Perhaps most importantly, we have learned that there are many unanswered questions and problems still to be addressed. We must determine if our interworking system can be extended to support all of the features recommended in the draft standard for interworking. The system proposed does not currently address issues such as type domains, security domains, and technology domains. More sophisticated and complex modelling of service information, particularly dynamic characteristics which have yet to be examined, would also be beneficial. It is also unclear how this system would perform at a larger scale – a detailed performance analysis of the current system and alternatives is clearly required. Another key issue is how compatible is this system to existing trader applications, and what must be done to provide an interworking mechanism that supports these applications. Finally, we must investigate and analyse other approaches to interworking to determine which is, in fact, the best one to take.

As more work is done in this area, these problems will be addressed and overcome. This work has laid the groundwork to help this evolving process and gained us valuable experience in this area. With our detailed design and our prototype implementation, the above issues can be examined as extensions and enhancements to the current system that we have developed. In the future, this system can be expanded to an interworking trader testbed for further experiments and investigations in this growing area of distributed computing.

About the Authors

Michael J. Katchabaw is a Ph.D. student in the Department of Computer Science at the University of Western Ontario. His research interests include distributed computing, network management, and distributed multimedia. Mr. Katchabaw can be reached at katchab@csd.uwo.ca.

Meeta M. Khurana is a Masters student in the Department of Computer Science at the University of Western Ontario. Her research interests include distributed systems, computer graphics, and ray-tracing and modelling techniques. Ms. Khurana can be reached at mmk@csd.uwo.ca.

James W. Hong is an assistant professor in the Department of Computer Science and Engineering at the Pohang University of Science and Technology. Dr. Hong received a Ph.D. from the University of Waterloo in 1991. His research interests include distributed computing, operating systems, software engineering, and network management. Dr. Hong can be reached at jwkhong@vision.postech.ac.kr.

Michael A. Bauer is chairman of the Department of Computer Science at the University of Western Ontario. Dr. Bauer received a Ph.D. in computer science from the University of Toronto. His research interests include distributed computing, distributed directories, and software engineering. Dr. Bauer can be reached at bauer@csd.uwo.ca.

References

[1] ANSA. *ANSA Reference Manual*. Release 1.01, APM Ltd., Cambridge UK, July 1989.

[2] M. Bearman and K. Raymond. Federating Traders: An ODP Adventure. *Proc. of the IFIP Workshop on Open Distributed Processing*, Berlin, Germany, 1991.

[3] M. Bearman and K. Raymond. Contexts, Views and Rules: An Integrated Approach to Trader Contexts. *Proc. of the International Conference on Open Distributed Processing*, pages 153–163, Berlin, Germany, September 1993.

[4] John Bloomer. *Power Programming with RPC*. O'Reilly & Associates, Inc,, Sebastopol, CA, 1992.

[5] CCITT. *The Directory - Overview of Concepts, Models and Services, CCITT X.500 Series Recommendations*. CCITT, December 1988.

[6] CCITT. *The Directory - Overview of Concepts, Models and Services, Draft CCITT X.500 Series Recommendations*. CCITT, December 1991.

[7] D. R. Cheriton and T. P. Mann. A Decentralized Naming Facility. *Technical Report 86-1098, Department of Computer Science, Stanford University*, February 1986.

[8] S. Dyer. The Hesiod Name Server. In *USENIX Winter Conference*, pages 183–189, Dallas, Texas, February 1988.

[9] J. Indulska, K. Raymond, and M. Bearman. A Type Management System for an ODP Trader. *Proc. of the International Conference on Open Distributed Processing*, pages 141–152, Berlin, Germany, September 1993.

[10] ITU-TS. *Basic Reference Model of Open Distributed Processing Part 4: Architectural Semantics*. ITU-TS Rec X.904, ISO/IEC 10746-4, August 1992.

[11] ITU-TS. *Basic Reference Model of Open Distributed Processing Part 1: Overview and Guide to the Use of the Reference Model*. ITU-TS Rec X.901, ISO/IEC 10746-1, July 1992.

[12] ITU-TS. *Draft ODP Trading Function*. ITU-TS SG7.Q16 Draft Recommendation, July 1994.

[13] ITU-TS. *Basic Reference Model of Open Distributed Processing Part 2: Descriptive Model*. ITU-TS Rec X.902, ISO/IEC 10746-2, November 1992.

[14] ITU-TS. *Basic Reference Model of Open Distributed Processing Part 3: Prescriptive Model*. ITU-TS Rec X.903, ISO/IEC 10746-3, November 1992.

[15] K. Muller-Jones, M. Merz, and W. Lemersdorf. The TRADEr: Integrating Trading Into DCE. *Proc. of the IFIP International Conference on Open Distributed Processing*, Brisbane, Australia, February 1995.

[16] OSF. *The OSF Distributed Computing Environment Rationale*. Open Software Foundation, Cambridge MA, May 1990.

[17] L. L. Peterson. The Profile Naming Service. *ACM Transactions on Computer Systems*, 6(4):341–364, 1988.

[18] C. Popien and B. Hager. The ODP Trader Functionality Applied to the Integrated Road Transport Environment. *Proc. of the Globecom'93*, pages 1202–1206, Houston, TX, November 1993.

[19] C. Popien and B. Meyer. Federating ODP Traders: An X.500 Approach. *Proc. of the ICC'93*, Geneva, Switzerland, May 1993.

[20] A W. Pratten, J. W. Hong, M. A. Bauer, H. Lutfiyya, and J. M. Bennett. A Resource Management System Based on the ODP Trader Concepts and X.500. *Proc. of the Fourth International Symposium on Integrated Network Management*, Santa Barbara CA, May 1995.

[21] P. Putter and J. D. Roos. Relationships: Implementing Transparency in Distributed Management Systems. *Proc. of the IEEE First International Workshop on Systems Management*, pages 118–124, Los Angeles, CA, April 1993.

[22] A. Vogel. Interworking Trader Project. Project documentation, available via HTTP from `http://www.dstc.edu.au/AU/projects/iwt/welcome.html`, May 1995.

[23] A. Vogel, M. Bearman, and A. Beitz. Enabling Interworking of Traders. *Proc. of the IFIP International Conference on Open Distributed Processing*, Brisbane, Australia, February 1995.