The following paper was originally published in the
Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)
Santa Fe, New Mexico, April 27-30, 1998

Dynamic Management of CORBA Trader Federation

Djamel Belaid, Nicolas Provenzano,
and Chantal Taconet
Institut National Des Telecommunications

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Dynamic Management of CORBA Trader Federation

Djamel Belaïd, Nicolas Provenzano, Chantal Taconet[1]
Institut National des Télécommunications
Evry, France

Abstract

In Wide Area Networks, tools for discovering objects that provide a given service, and for choosing one out of many are essential. The CORBA trading service is one of these tools. A trader federation extends the limit of a discovery, thanks to the cooperation of several trading servers. However, in a federation, cooperation links are manually and statically established. In this article, we propose an Extended Trading Service, which manages a trader federation dynamically. With Extended Trading Service, optimized links between traders are automatically set up thanks to the use of a minimum-weight spanning tree. Cycles in discovery propagation are eliminated. Links evolve dynamically in order to adapt to the modification of the underlying topology and the failure of an intermediate trader or link. The Extended Trading Service is able to organize discovered objects from the nearest to the furthest according to a distance function chosen for the federation. Such management is provided by specialized trading servers conforming to OMG Trading Service Specification.

1. Introduction

The expansion of Wide Area Networks (**WANs)** has already led to information superhighways. From now on, a huge number of computer services are being developed and made available on WANs. These services should be available from a wide range of computer stations and through a wide range of networks. Object middleware, such as those built with *CORBA* (Common Object Request Broker Architecture) [OMG97], help to implement distributed services without taking care of distribution configuration.

In WANs, huge number of clients, and distances between clients, have naturally led to replicate some server objects on geographically distributed networks. Every day, new replicas may appear. One issue is to offer final users tools for transparent discovering services and in the case of replicated servers, for discovering the "best" one, which may be different for each client.

Tools currently offered to find out server objects are not adapted for finding the "best" server for each client. Traditional name servers such as *DNS* [Mock87], compel one to give different names to different replicas (e.g. different *URLs* [Bern94]) and so don't help end users to choose the best server. The *DNS* support for replication [Bris95] automatically selects a server through a round robin algorithm and as a result the chosen server is not adapted to each client. Discovery tools, such as *Alta Vista Search Engine* [Seit96], offer global searching on the Internet but selection is on text information only.

Some new kinds of discovery tools are appearing. With the *Globe* location service, servers are registered in a global hierarchical tree [vS96], which is then used for finding the nearest server. The Internet community is thinking of *URN* (*Uniform Resource Name)* [Moat97] to replace *URL* in order to have better support for replicated and movable servers. The trading service specification, proposed for *ODP* [ODP93] and then *CORBA* [OMG96], has been defined to find the best server(s) for each client thanks to a service type and a list of properties.

In WANs, some services are available thanks to the cooperation of a set of distributed servers. We illustrate this feature through the following examples. The *News USENET* [Kant86] is distributed to final users thanks to the cooperation of several news servers distributed on the Internet. In the *MBone* [Deer90], the

[1] INT, 9 rue Charles Fourier, 91 011 Evry Cedex, France.
{Djamel.Belaid, Nicolas.Provenzano, Chantal.Taconet}@int-evry.fr

subset of Internet supporting multicast routing, packets are forwarded thanks to tunnels set up between a set of cooperating multicast routers. And a federation of cooperating traders may offer the trading service. The study of these examples shows that cooperation links between these servers are set up manually by human administrators on each server. These cooperation links are neither necessarily adapted to the underlying network topology nor fault tolerant. One issue is to offer a tool for linking cooperating servers efficiently and dynamically.

The *CORBA* trading specification does not offer any tool for linking cooperating traders dynamically either. In this article, we define a specialized trader, conforming to the *OMG* specification, which offers dynamic management of trader federation. Our federation is based on the cooperating server graph model [Taco97] that optimizes the links set up between cooperating traders and helps them to find the nearest server to each client.

This article is organized as follows. In Section 2, we present a synthesis of the *CORBA* trading service specification and we study limitations of this service in the WAN context. In Section 3, we summarize the Cooperating Server Graph model that offers to manage links between cooperating servers dynamically. We use this model in Section 4 to define the architecture of a specialized trader for WANs. In Section 5 we study its implementation and compare its behavior with traditional traders in the WAN context.

2. CORBA Trading Service description

In this section, after a brief description of the *CORBA* architecture, we present the main features of the CORBA Trading Service, and then some optimizations of this service intended to the WAN context.

2.1. CORBA Architecture

Common Object Request Broker Architecture (*CORBA*) [OMG97] is an open distributed object computing infrastructure standardized by the *Object Management Group* (*OMG*). *CORBA* allows development of applications in which distributed objects communicate with one another thanks to well-defined interfaces, no matter where objects are located or how objects are implemented.

In *CORBA* each object is identified by an **object reference** and is associated to an interface and an implementation. An interface allows clients to access a set of services offered by a server object. Interfaces are described with the *CORBA Interface Description Language* (*CORBA-IDL*).

CORBA Architecture consists of the following components:

- **Object Request Broker** (*ORB*) is a middleware that establishes client-server interaction between distributed objects. When a client invokes a method on a server, whatever programming language or operating system used for server implementation, *ORB* has to find the server implementation location, deliver the request to this server, and return invocation results to the client. In order to allow interaction between objects on different *ORBs*, *OMG* has defined a *General Inter-ORB Protocol* (*GIOP*), which specifies a standard transfer syntax and protocol. *GIOP* is designed to operate over a connection-oriented transport protocol. *Internet Inter-ORB Protocol* (*IIOP*) is a concrete implementation of the abstract *GIOP* for *TCP/IP*.

- **Object Services**, is a collection of services that provide basic, nearly system-level, functions for implementing objects. We can mention Naming Service, Life Cycle Service, and Trading Service.

- **Common Facilities**, is a collection of services shared by many applications. For example, graphical objects may be used by every application for providing a user interface.

- **Applications Objects** are specific to each application. They may use any *CORBA* objects (e.g. Object Services and Common Facilities). They are not standardized by *OMG*.

2.2. Obtaining an object reference

In order to invoke a method on a server object, a client must first hold an **object reference** to this server. An object reference is associated to at most one CORBA object. With an object reference, the *ORB* is in charge

of locating the object and delivering the invocation to the server.

Object references may be obtained by the following means. First, with `resolve_initial_references` *ORB* operation, clients may obtain references to well known services such as `InterfaceRepository` and `NameService`. Clients may also obtain object references from output parameters of any method. The Naming Service allows clients to obtain object references thanks to object symbolic names. And finally, the Trading Service allows clients to get object references selected thanks to a type of service name and a list of properties.

2.3. Trading Service

In this subsection we describe the main features of the trading service.

The trading service has been designed to allow the registration and discovery of objects. A **trader** is an object that provides the trading service in a distributed environment. Server objects advertise or **export** their service offers to traders. **Exporters** may be server objects or other objects acting on the behalf of the server. Client objects invoke traders to discover or **import** service offers matching a given type of service and a set of **properties**. Clients are called **importers**; they can be the consumers of the service or act on behalf of other objects.

A **service type** is defined in a Service Repository with an interface type and a set of zero or more properties. Each property is described by a name, a mode and a type of value. If a property mode is mandatory, then each instance of the service type must provide an appropriate value for this property when exporting its service offer. Each service type is identified by a unique `ServiceTypeName`.

A **service offer** consists of a `ServiceTypeName`, a list of properties (property name and value), and an object reference to the interface providing the service. Some properties are dynamic; for these, values are not in the service offer, but obtained explicitly from the interface of a dynamic property evaluator given by the exporter of the service.

The main trader interfaces are shown in Figure 2.1. The most important ones are the `Register` and the `Lookup` interfaces. The `Register` interface provides the `export` method for exporting a service offer. The `Lookup` Interface provides the `query` method for importing a list of service offers. The importer may express its **preferences** with a constraint language. Traders organize discovered service offers according to these preferences.

Traders can be linked together in a **trader federation**. Figure 2.1. gives a simple example of federation between traders *A*, *B*, and *C*. Target trader (e.g. trader *B*) establishes a link with a source trader (e.g. trader *A*) using the `Link` interface. Then, the source trader (i.e. trader A) is able to invoke target trader (i.e. trader B) with a `query` method. These links are therefore explicitly created and are unidirectional. All the links form a directed graph called the **trading graph**. The `Link` interface provides methods to manage the links, such as the `add_link` method used by a target trader to define a new link to a source trader, and the `remove_link` method to remove a link.

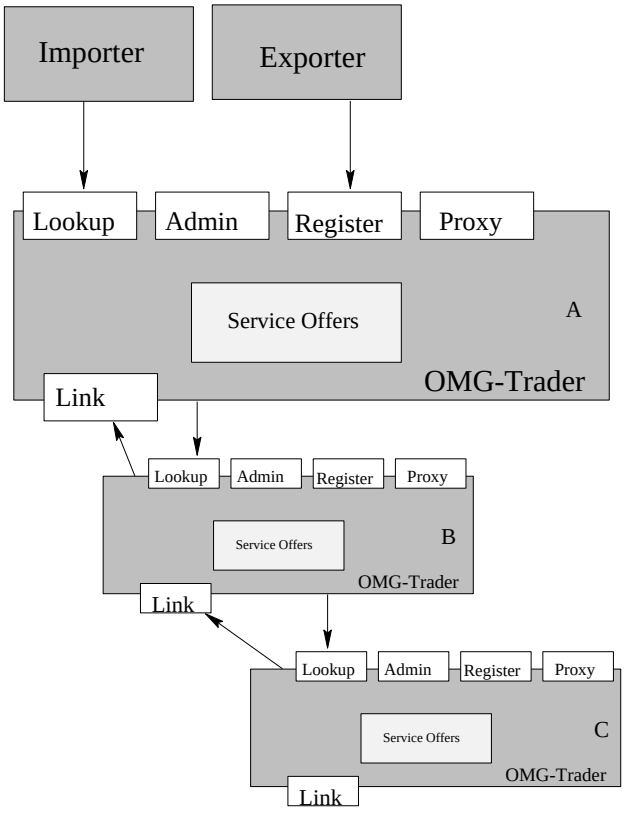

Figure 2.1: Traders Interfaces and Federation

A federation allows traders to extend an importation to a trading graph. **Importation policies** modify trader behavior for a discovery in a federation. Importation

policies are associated to each trader, each link, and each importation. A combination of these policies conditions the list of traders visited for an importation. Trader policy overrides link policy, which itself overrides importer policy. The main importation policies are given as follows: *(i)* `search_card`, gives the number of service offers to be searched; *(ii)* `return_card`, gives the number of ordered service offers to be returned to the client; *(iii)* `hop_count`, gives the maximum number of links that may be visited for a search; *(iv)* `starting_trader`, gives a path to a remote trader on which the search must start; *(v)* `follow_policy`, defines the trader behavior for propagating a search. The following policies are: *(i)* `local_only`, only locally registered service offers are returned; *(ii)* `if_no_local`, the search is only propagated if the number of local offers matching the request is less than the number of offers to be returned (i.e. `return_card`); *(iii)* `always`, the search is propagated till the expected number of offers is reached (i.e. `search_card`).

Besides the `Register`, `Lookup` and `Link` interfaces, the trading service defines three other interfaces. The `Admin` interface allows one to modify and list interfaces and policies supported by a trader. The `Proxy` interface is used to register service offers for which the object reference is not known at the exportation but obtained at query time thanks to a proxy object. And the `ServiceTypeRepository` interface is used for the management of the repository service types. Traders have to implement at least the `Lookup` interface.

2.4. Trading service optimization

In this sub-section, we present possible optimizations of *OMG* trading service and especially for that particular part which concerns trader federation.

Because of the huge number of services available on WANs, it is easier for end users to search a service according to its characteristics rather than to its name. Indeed, it's easier to get information on Internet through a search engine than by giving its URLs. Because of the increase in the number of services, the trading service is bound to become more useful than the naming service.

Trader federation is an interesting feature in the context of WANs: it allows one to distribute the trading service on several traders. However, improvement could be achieved on the following points.

As they are now defined, trader federations are bound to be established "manually" by an administrator. As a result, they may not be adapted to the underlying network topology. Furthermore, trader graphs may contain cycles (i.e. a search may visit the same trader several times).

As the federation is usually static, it cannot react in a transparent way to network events such as trader or communication link failures, or changes on the underlying network topology. Therefore, it doesn't adapt to events occurring on the underlying WAN.

The trading service doesn't define the concept of distance between objects. Consequently, the client cannot express the search of the nearest service, and the trader cannot organize service offers according to distance between clients and servers, even though this information could be important because of differences in communication costs in a WAN.

The integration of the *Cooperating Server Graph* model, which we describe in Section 3, in the trading service bring solutions to the above remarks and therefore would improve and optimize this service. We present in Section 4 a proposal of an *Extended Trading Service* using this model.

3. Cooperating Server Graph model

In this section, we summarize the *Cooperating Server Graph* model (**CSG**), which is described in details in [Taco97b]. The aim of the CSG model is to optimize and dynamically manage links between cooperating servers over a WAN. This model defines a protocol for dynamically updating the links according to different events happening, either to some servers, or to the underlying WAN. This model has already been adapted for the cooperation of WAN location servers for the *Chorus* micro-kernel [Taco97a]. We present in Section 4 the use of this model for dynamically managing a trader federation.

3.1. Cooperating Server Graph definition

On a WAN, we consider a set of **cooperating servers** (several hundred or so) which cooperate in order to

offer a service (e.g. a federation of traders which offers the trading service). The cooperating servers are geographically distributed on different computer **sites** (e.g. LANs) which may be separated by long distances. Each site is made up of physically close machines. The sites are logically linked (e.g. they belong to the same company or cooperate for a given project) and physically connected by an underlying WAN.

As there may be several sets of cooperating servers on the same WAN, we associate to each one a unique identifier (e.g. symbolic name).

The model uses a **distance** function. At a given time, this function associates a value to each couple of cooperating servers. This value has to be representative of the communication cost between the couple of cooperating servers (e.g. financial cost, latency induced by physical distance or available bandwidth), and may change over time. For Internet, we have used the distance function used by routing protocols (i.e. number of hops between sites).

We build a graph, namely a CSG, in which the nodes are the cooperating servers. The nodes are linked by a weighted edge providing that a distance value has been evaluated between the two nodes. The CSG is simple, k-connected and not oriented.

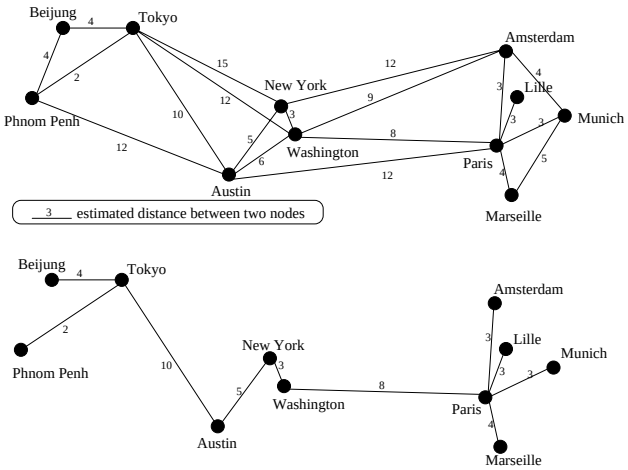

Figure 3.1: A CSG example (a) and its associated broadcast tree (b)

We assume that the number of nodes in a CSG is low (some hundreds or so). So each cooperating server stores in its own memory the layout of the CSG. Thanks to the CSG, all the cooperating servers calculate the same **broadcast tree** between all the nodes. We use a minimum-weight spanning tree calculated by the Prim algorithm [Prim57]. This tree is used to broadcast information to all the cooperating servers. With the broadcast tree, broadcasting is made with a minimum communication cost (according to the distance function), with a distribution of the communication load between all the nodes and without the need of a stop control for eliminating the CSG cycles.

Figure 3.1-a gives a simple example of a CSG with eleven cooperating servers distributed all over the world. Figure 3.1-b shows the broadcast tree calculated for this configuration.

3.2. Dynamic update of the CSG

The model includes a protocol for updating the CSG and its associated broadcast tree in order to take into account the following events: the addition or removal of a cooperating server, modification of the underlying WAN topology which leads to some CSG distance changes and temporary failure of a cooperating server or of a communication link.

In order to be more efficient and because of differences in the duration of events, the model offers three levels for taking those events into account: *(i)* the alternative behavior in case of failure; *(ii)* the local modifications; *(iii)* and the global change of CSG version.

When a failure is just discovered, i.e. when one node can't propagate an information to its neighbors in the tree, this node uses the alternative behavior in case of failure. It propagates the information on behalf of the failed neighbor[2] to the neighbors of the failed neighbor in the tree. For example, in Figure 3.1, if node *Phnom Penh* cannot propagate to node *Tokyo*, node *Phnom Penh* will decide to propagate to nodes *Beijung* and *Austin* on behalf of node *Tokyo*. This behavior is possible because of the global knowledge of the broadcast tree. This behavior maintains the continuity of the service.

The local modification level is used for long time failure, long time failure recovery, addition and removal

[2] *In order to simplify, we call it the failed neighbor but the communication failure may come from a failed server or from a network failure.*

of cooperating servers. A local modification consists in a coherent change of the broadcast tree seen on a node, its neighbors, and the neighbors of its neighbors. For example, if the failure of node *Tokyo* lasts after a given delay (e.g. several minutes), a new configuration of the tree shown on Figure 3.2 between nodes *Beijung*, *Phnom Penh* and *Austin* is calculated. This modification concerning node *Tokyo* is made coherently on *Tokyo*'s neighbors in the broadcast tree (i.e. *Beijung*, *Phnom Penh* and *Austin*) and on the neighbors of its neighbors broadcast tree (i.e. *New York*). Local modifications keep a broadcast tree, but this tree is no longer a minimum-weight spanning tree.

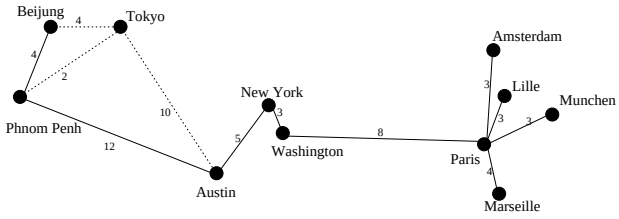

Figure 3.2: Local modification of the broadcast tree

A CSG version change is activated as soon as the degradation rate[3] of the broadcast tree goes past a given threshold. The activation is triggered by a Changing Version Server (**CVS**) chosen dynamically in the set of cooperating servers. The new version takes into account all the events considered as permanent since the last version: very long time failures, very long time distance changes, addition and removal of nodes. The new version is propagated on the broadcast tree. Two nodes have to agree on a version before they can communicate.

All the CSG updates are made dynamically. Thanks to the three update levels, the number of version changes is reduced. The links between all the nodes follow the evolutions of a CSG and its underlying WAN topology. The model tolerates a great number of failures. In case of too many failures leading to dividing the CSG into several isolated classes[4], server cooperation is limited inside each class.

We present in Section 4 how the integration of this general model optimizes a *CORBA* Trader federation.

4. The Extended Trading Service

4.1. Global description

The Extended Trading Service is an evolution of the OMG trading service, which integrates the Cooperating Server Graph model (cf. Section 3). The aim of this evolution is to optimize the management of trader federations and object importation over a set of traders scattered on a WAN.

The Extended Trading Service is offered by a federation of CSG-traders belonging to the same logical domain. A CSG-trader is a specialization of an OMG-trader preserving OMG-trader interfaces. For an importer or an exporter, the CSG-trader is therefore entirely conform to the OMG specification and offers the same service as the OMG-trader. Any implementation of the OMG trading service may be specialized in a CSG-trader.

We consider a CSG in which each node is a CSG-trader. Besides its OMG trading service function, a CSG-trader ensures automatic federation of CSG-traders. Each CSG-trader stores its `CSG` and calculates the broadcast tree. In a CSG-traders' federation, the propagation of importations follow the broadcast tree. Every CSG-trader manages dynamically (in collaboration with the other CSG-traders) the links of the federation. Links evolve according to events occurring on the network. CSG-traders may be linked to OMG-traders using OMG links. Object search is then performed according to client choice either in OMG mode using OMG links or in CSG mode using the broadcast tree. Finally, in order to reduce the number of extended importation, each CSG-trader manages a cache of service offers.

4.2. The CSG-trader architecture

The general architecture of a CSG-trader is presented in Figure 4.1. A CSG-trader consists of one OMG-

[3] The degradation rate is estimated with the sum of the weights of the degraded broadcast tree and the sum of the weights of the minimum spanning tree that could be used.

[4] If a node cannot communicate with a node and some of the neighbors of the failed neighbor it assumes there is a partition. Its class is made up with the sub-trees with which the communication is still possible.

trader part, called the trader part, and a CSG specialization called the CSG part.

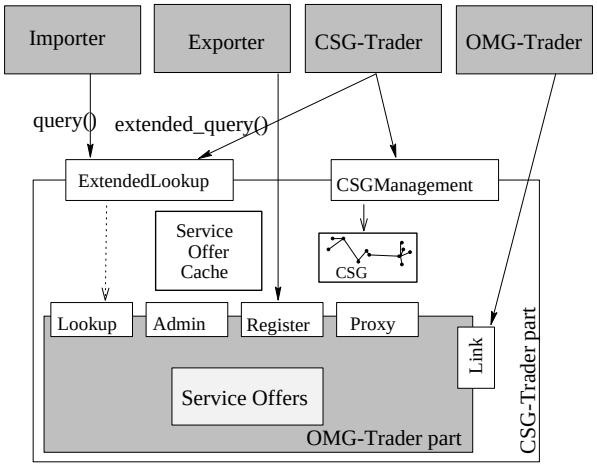

Figure 4.1: CSG-trader Architecture

4.2.1. The OMG-trader part

As a CSG-trader is derived from an OMG-trader, it inherits from all the OMG-trader interfaces namely the `Lookup` interface and possibly the `Register`, `Admin`, `Proxy` and `Link` interfaces. The `Link` interface is only used to create links between OMG-traders and CSG-traders, which we call OMG links. These links and their follow policies are managed by the trader part. Because links between CSG-traders have a different semantic they are managed by a special interface.

4.2.2. The CSG-trader federation

A CSG-trader federation is established according to the CSG. The links between CSG-traders, which we call CSG links, are entirely managed by the CSG part and are not explicitly created. Indeed, every CSG-trader knows all others CSG-traders, calculates a broadcast tree, and consequently knows its neighbors in the broadcast tree. A CSG link is essentially composed of the target CSG-trader name as well as a reference to its interfaces.

4.2.3. The CSG-trader specific interfaces

The extended trading service inherits interfaces from the *OMG* trading service. We add two new interfaces: the `CSGManagement` interface and the `ExtendedLookup` interface.

The `CSGManagement` interface provides methods for the management of a CSG. Among them we can mention `ask_for_csg` that allows a new CSG-trader to get the `CSG`, in order to set up a link with the nearest CSG-trader using the `add_extended_link` method.

The `ExtendedLookup` interface is a derivation of the OMG-trader `Lookup` interface. It provides several methods: *(i)* the overrided `query` for all clients (except CSG-traders); *(ii)* the `extended_query` for the propagation of a search over a CSG-trader federation; *(iii)* the `extended_answer` to return the result of a search to the source trader. Compared to the `query` method, the `extended_query` add two parameters: a `CSG` identifier and the name of the CSG-trader initiator of the importation. It is invoked in a one way method. The service offer result (if any), as well as the reference of the CSG-trader that gives the offer, is returned directly to the initiator CSG-trader later, using the `extended_answer` one way method. The initiator trader is then able to evaluate the distances of each discovered offer.

4.2.4. Service importation in a CSG-trader federation

In the CSG mode, a search is limited in a CSG. In order to allow clients to specify a `CSG` identifier, we define the `CSG` property. The `CSG` property is useful only for the CSG part of a CSG-trader.

When a CSG-trader receives a query request with its `CSG` identifier, it firstly asks its trader part with `local_only` policy (without the `CSG` property), and then, if necessary, propagates the request. If the CSG-trader does not belong to the indicated `CSG`, it can be considered as a **relay trader** and forward the request to a CSG-trader belonging to the required `CSG`. If the client doesn't indicate any `CSG` identifier, then the search is not carried out over the CSG federation but is accomplished following the OMG links exclusively.

4.2.5. Service offer cache

The goal of the CSG-trader is to optimize importation over a WAN. For this purpose, each CSG-trader stores a service offer cache in which it stores results of previous extended importations. All clients of the same CSG-trader benefit from this cache.

The service offer cache holds an LRU table in which each cell consists of a service type name, a set of properties, the policy used to discover this offer, and the reference of the CSG-trader that returned this service

offer. When a CSG-trader receives a `query` request it looks in the cache for a cell matching the `query`. If it finds any, it sends a "`local_only`" `query` request to the CSG-trader referenced in that cell in order to verify the validity of the service offer and get its dynamic properties. If the target CSG-trader returns the service offer, the cell age is updated; otherwise the cell is deleted.

4.2.6. Importation policies

The CSG-trader provides the same importation policies as the OMG-trader. We present here the importation policies whose semantic has been adapted to CSG federations.

With the `if_no_local` and `always` policies, the client request is propagated following the broadcast tree. Each intermediate CSG-trader invokes the one way `extended_query` method in parallel to all the following sub-trees. Results, if any, are returned directly to the initiator CSG-trader. With this behavior, the number of intermediate traders waiting for answers is significantly reduced, the drawback is that discovery goes on on each subtree independantly even if results have been found on other subtrees.

The `hop_count` policy preserves its semantic and applies to the broadcast tree. For example, with a `hop_count` of "1", only the initiator CSG-trader's neighbors on the tree will be visited.

Only the `if_no_local` policy uses the cache. In order to ensure the locality of the service offers, the cache is not used with the `local_only` policy. We also have chosen not to use the cache with the `always` policy, in order to preserve the quality of the results rather than the performance of the search.

Finally, if every server object exports its service offer to the nearest CSG-trader, and the client imports from the nearest one, the extended trading service may organize the results from the nearest to the furthest, thanks to the use of the CSG graph.

5. CSG-trader implementation

In this section, we first present the representation of a `CSG` in a CSG-trader, we then describe our CSG-trader prototype, finally we compare the OMG trading service and the Extended Trading Service with a simple federation example,.

5.1. Representation of a CSG

A CSG-trader federation is represented on each CSG-trader with the same data structure, in which each node or CSG-trader is described by a `TraderElement` which consists of the following components.

- The CSG-trader identification in the CSG.
- The distance function type (adapted to the federation).
- The reference of its `GSCManagement` interface (for dynamic CSG evolution).
- The reference of its `ExtendedLookup` interface (for query operation propagation).
- Its network address (for evaluation of distances between CSG-traders).
- The sequence of its neighbors in the broadcast tree (this information represents the broadcast tree).

A CSG is represented by a *CORBA* object (type `CSG`). In this object is stored: *(i)* the running version of the CSG (global information common to all nodes), and *(ii)* node specific information for recording local modifications. The different components of this object are as follows.

- `CSG` identifier
- The running CSG version number
- The number of CSG-traders (known locally). Because of local modification unknown on this node, this number may be different from the number of CSG-traders in the current broadcast tree.
- A sequence of `TraderElement`. This sequence may not be the same on each node because of local modifications.
- The distance matrix (distance between CSG-traders). If a distance has not been evaluated, the infinite value is attributed.
- The table of distance between the local CSG-trader and other CSG-traders**.** If the distance is infinite in the matrix, the distance is evaluated by the sum of distances in the shortest path between the two nodes (the CSG is connected).

A `CSG` is described by an *IDL* interface and may be obtained by another CSG-trader. This feature is interesting for the addition of a new CSG-trader in the CSG, and for the propagation of a `query` request to another CSG.

5.2. Prototype

We have implemented a prototype of CSG-trader on *Orbix 2.1*[5] with *Sun Solaris 2.5*. This prototype uses *IIOP* references.

At the time of our implementation we did not have the sources of any OMG-Trader, so we have implemented a **simplified OMG-trader** which supports `Lookup`, `Register` and `Link` interfaces. But we can easily adapt our prototype to any OMG-trader implementation.

Our CSG-trader prototype is a specialization of the *simplified OMG-Trader*, which furthermore implements `CSGManagement` and `ExtendedLookup` interfaces, provides dynamic updates of the CSG and manages the *offer service cache*.

With this prototype we have done the following elementary tests on a LAN.

- Addition and removal of a CSG-trader in the federation.
- Automatic CSG version change on all the nodes.
- Propagation of importation on a CSG-trader federation with `local_only`, `if_no_local` and `always` policies.
- The alternative behavior in case of failure of an intermediate CSG-trader.
- CSG-trader and OMG-Trader cohabitation.

5.3. Comparison between a CSG-trader and an OMG-Trader

In order to give an interesting comparison between a CSG-trader and an OMG-Trader we would have needed a complete *OMG-trader* implementation and a testbed for WANs. We did not have any of these two conditions, that is the reason why we don't give any performance comparison. We present here a comparison illustrated by an example in order to highlight interesting features of the Extended Trading Service.

In this section, we use the CSG federation of Figure 3.1. An example of possible associated *OMG* trading graph is given in Figure 5.1-a. In Figure 5.1 unidirectional OMG links are represented with one arrow, other links are bi-directional. In this figure we present the invocations needed for a discovery in the OMG graph (Figure 5.1-a) and in the CSG federation (Figure 5.1-b).

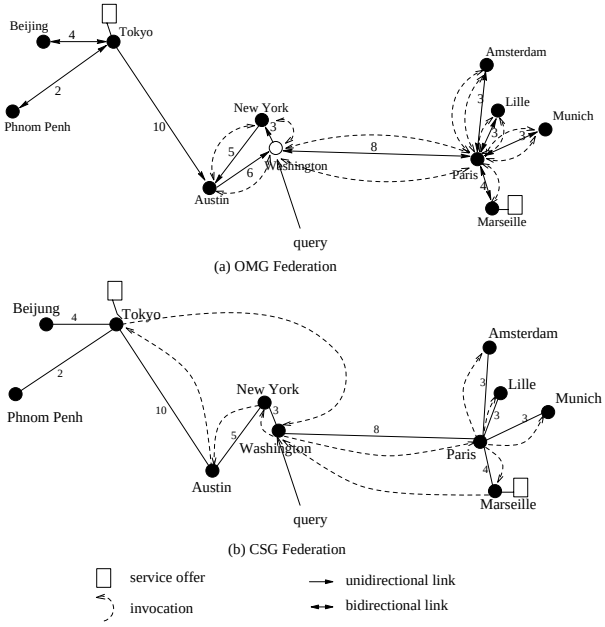

Figure 5.1: Example of discovery

For this example, we compare the number of method invocations needed for OMG Trading Service and Extended Trading Service. The query request example uses the policy `if_no_local`, the `search_card` and the `return_card` both have value 1, the initiator trader is *Washington*. The matching service offers are on nodes *Tokyo* and *Marseille*. The Extended Trading request propagates requests in parallel on all the following sub-trees. Dashed arrows symbolize method invocations.

For this example, an importation on OMG federation generates 12 two-way method invocations and on the CSG-federation it generates only 10 one way method invocations. In OMG federations, double links (bi-directional links) lead to double invocations (even though there is a stop control). So, even if the OMG-trader graph is a tree, two times more invocations would be needed. Number of invocations will also be reduced by the CSG trader cache management.

In order to avoid cycles, the OMG-traders need to store and compare request identifiers (case of cycle *Washington*, *New York* and *Austin*) at each node.

[5] Iona Technology *CORBA implementation*

With an OMG-trader federation, no guarantee is given on the existence of a path between each pair of nodes. In the example, the *Tokyo* service offer can't be found. Special attention is needed to configure an OMG-trader federation.

With an OMG trader federation each intermediate trader in the importation has to be waiting for an answer (RPC invocation). With the CSG federation only the initiator trader is waiting for an answer.

We argue that CSG-traders would facilitate trader federation. However, more tests are needed to verify the efficiency of the overall CSG federation mechanisms.

6. Conclusion

Because of distributed computing, the evolution and diversity of services offered on today's and tomorrow's WANs, discovering tools such as the trading service should become essential for end users.

In this paper, we have described the *CORBA* trading service specified by the *OMG*. With this service clients may import service offers exported on traders. Cooperating traders may be federated to offer extended searches. Yet, we have shown that as links are statically and manually established they are not adapted to the underlying network topology and do not evolve dynamically. Moreover, they do not help clients to choose the nearest replicated object, while, because of communication delay and cost on WAN; this would be an important feature.

We have presented the *Cooperating Server Graph* model. With this model, the links between cooperating servers are established dynamically. Furthermore, thanks to an inter server protocol, the links evolve to react to different events such as intermediate servers or communication links failures, and modifications in the underlying network topology. With this model, the propagation of information to all servers is efficient. The knowledge of distance information between the servers allows traders to organize the results from the nearest to the furthest.

We have defined the CSG-trader that integrates the CSG model in an OMG-Trader. CSG-traders offers the following optimizations. Trader federation is established and evolves dynamically. Extended service offers searches follow a minimum-weight spanning tree. Assuming that importation and exportation are sent to the nearest trader to clients and servers, searches may find the nearest server to each client. Asynchronous treatment of requests increases the number of requests handled in parallel by each trader.

We have then described a CSG-trader prototype for *Orbix* 2.1, which is a specialization of a simplified trader that we have implemented.

In the definition and implementation of the CSG-trader we paid a special attention to stay conform to the trading service interface specification. Our optimizations are transparent for trading service clients. In order to facilitate the choice of a search domain (i.e. a CSG), and of a starting trader, it would be interesting to adapt the trader specification.

Trading service and migration

We would like to emphasize that migration, taken into account in *CORBA* life cycle service, and trading exportation should be linked together. In *CORBA* specification, an object reference should stay valid after a migration. So, a service offer stays valid after a migration. Yet, in order to both facilitate *ORB* location service and preserve the nearest server semantic offered by CSG-traders in case of object migration, we argue that a server migration should be coupled with the migration of its associated service offer. And so service offers will be registered on the trader which is the nearest to the server object.

CORBA domains

CORBA specification defines several notion of domains (interoperability domains, policy domains, security domains). A more precise definition of administrative domain seems to be an important issue.

Just like CSG, a domain may take into account the logical relationship between *ORBs*. For example, all the computer sites of a company may define a *CORBA* domain. Some of the *CORBA* services could benefit from such domain definition. The life cycle service could limit some migration and replication inside a *CORBA* domain. The trading service could restrict searches inside a *CORBA* domain. Every object would be associated to one or several administration domains. And so, some operations may be authorized

between objects of the same domain only, while others may be authorized between different domains.

References

[Bern94] T. Berners-Lee, L. Masinter, M. McCahill. Uniform Resource Locators (URL), *RFC1738*, December 1994.

[Bris95] T. Brisco. DNS Support for Load Balancing. RFC 1794, April 1995.

[Deer90] E.S. Deering and D.R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM transactions on Computer Systems*, 8(2), May 1990.

[Moat97] R. Moats. URN Syntax. *RFC 2141*, May 1997.

[Mock87] P. Mockapetris. Domain Names Implementation and Specification. *RFC1035*, November 1987.

[ODP93] Information Technology- Open Distributed Computing – ODP Trading Function. *ISO/IEC JTC1/SC21.59 Draft, ITU-TS-SG 7 Q16 rapport*, November 1997.

[OMG96] Trading Object Service. *OMG Document 96-05-06, RFP5 submission*, May 1996.

[OMG97] Common Object Request Broker: Architecture and Specification. Revision 2.1 *OMG Document*, August 1997.

[Prim57] R.C. Prim. Shortest Connection Networks and some Generalizations. *Bell Syst. Techno. J*. 36, 1957.

[Seit96] R. Seitzer, E.J. Ray, and D.S. Ray. Alta Vista Search Revolution: How to find anything on the Internet. *Digital Press*, New Jersey, 1996.

[Taco97a] C. Taconet and G. Bernard. Object Location in Wide Area Networks. In *Porceedings of ERSADS'97 European Research Seminar on Advances in Distributed Systems*, Zinal, Switzerland, March 1997.

[Taco97b] C. Taconet. Graphe de Réseaux Coopérants et Localisation Dynamique pour les Systèmes Répartis sur Réseaux Etendus. *Ph.D. Thesis* University of Evry, France, October 1997.

[VS96] M. Van Steen, F.J. Hauck, and A.S. Tanenbaum. A model for World wide Tracking of Distributed Objects. In *proceedings of TINA'96*, Heidelberg, Germany, September 1996.