Achieving Interoperability through ODP Trading Function

Lea Kutvonen
Department of Computer Science
University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland

Abstract

Interoperability between autonomous decentralized systems needs support of an information service that can provide dynamic knowledge about available service providers within the network. In this paper, we describe trading service. Trading is an essential infrastructure function indicated by ODP (Open Distributed Processing) reference model, a metamodel of heterogeneous distributed processing. The purpose of trading is to support location transparency of services: it hides the location of service providers, even if they are located in separate, autonomous domains. Cooperation between trading service providers (traders) supports federation transparency, i.e. hides differences of technology used in separate domains. In cooperation, the traders have to mediate service type differences, negotiate policy differences and tolerate technology differences between domains. Together with other fundamental functions trading offers a platform for open distributed computing between autonomous subsystems.

1 Introduction

ODP (Open Distributed Processing) is a framework for systems that support heterogeneous distributed processing [5], [10]. This framework has been under development since the late 1980's within ISO and ITU as a joint effort. The standardization effort produces a reference model that organizes the discussion about distributed systems. It supports the design process of new standards and real systems by dividing the complex tasks into viewpoints. It also defines new service functions to be used within the viewpoints.

The goal of the ODP framework is to *provide a distribution transparent environment* for software components to interwork, despite they are supported by heterogeneous, autonomous platforms. Achievement of distribution transparency requires hiding of failures, locations and implementation technologies of service providers.

In the ODP framework, software components (and hardware, as well) are modeled as objects that request services (defined actions) from their environment (other objects). These requests, interactions, take place via interfaces. An object can initiate operations at another object by binding itself into an interface at the target object. Objects willing to interact may be located at distinct nodes, they may use different representation of operations and they may be introduced to the global system independently. For example, in a system, two authors might work on a shared document with differing tools, each from their own computers in their organizational networks. The considered objects in this example are the editing tools and the shared document file itself. Operations offered at the document object interface are reading, writing and locking of the whole document or well defined parts of it. Other examples of objects are representations of telephone switches, operating systems, and employees.

The key problems are, how the objects know, where the possible interfaces are located, how to choose between them, and how to transform between different task representations. Especially in decentralized environments, a supporting service is needed for objects to locate each other. The objects need a logically centralized location at which they may request information about the available interfaces. This supporting service is called the *trading function*. It is a general tool for flexible discovery of targets in a dynamic network environment. Targets are addressed by property descriptions that constrain the set of potential targets to a set that contains only the most suitable ones. The trading service can be considered as an advanced directory service which allows attribute-based search. The attributes can be evaluated at search time, and hence, the trader is able to check the availability or reachability of each purported target.

Trading service facilitates late binding of objects. Late binding means that an initiating object may invoke an operation at another object, whose identity or location it did not know before the invocation actually happened. An object can even use the services of an object created later than the initiating object itself. Late binding has the benefit of allowing flexible system evolution. It also helps to achieve improved availability and fault tolerance within the system, as objects may dynamically change the service provider they choose to interact with. For instance, if the client has statically bound itself to a name server or to a research database in a computer A that fails, it can not work until the computer A is brought up again. However, if the client utilizes the trading services, it can rebind itself to a similar server in a computer B instead, and obtain the required information there.

The foreseen usage of the trading function in the ODP framework is to support inter-object communication via interrogations and announcements [5]. The

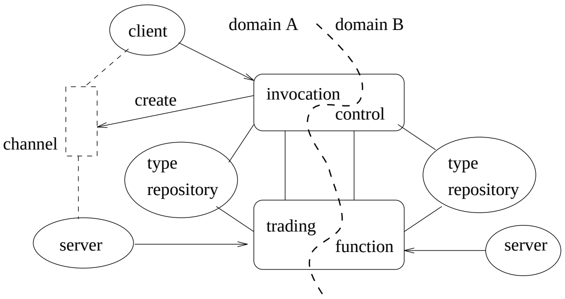

Figure 1: Communication architecture.

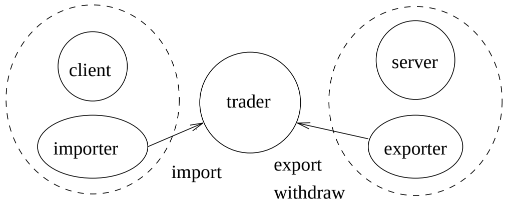

Figure 2: Trading community.

communication methods require, besides the trading function, also the support of type repository, channel creation (binding) and service invocation functions. The type repository function stores information about abstract service types and their concrete representation forms, in order to support translations between the service representations. The binding function sets up a channel between the objects. It requires information from the trading function and the type repository. When the binding has been established, the invocation controller can activate a service request on behalf of the client. The communication scheme is illustrated in Figure 1.

Within the ISO standardization group, the trading function is currently about to reach the committee draft status and thus stabilizing [6]. The DRYAD project, working at the University of Helsinki, has intensively followed the ODP standardization and has adopted the general goal and framework of the ODP model. The DRYAD project works on a design and experimental software for interoperable object invocations between autonomous systems [9]. The experimental work has created feedback for the standardization work.

The views given in this paper reflect both the ODP trading function specification and the DRYAD implementation of the corresponding functionality. There are no major logical differences between the ODP trading function and the DRYAD implementation. However, we see a difference in the intended dynamicity of the trader cooperation constellations. In the DRYAD project, we consider that the trading service is not limited to any specific object technology or any specific application area. It offers a general mechanism to match client requirements with server abilities.

We have so far introduced the purpose and the potential usage of the trading function within the ODP framework. In the following chapters, we describe a logical trader object using the ODP concepts of contracts and templates, and show how trading utilizes other ODP infrastructure functions. In this description, we follow the intentions of the ODP trading function, but have selected stronger and more restrictive ODP terms than have been accepted into the current committee draft proposal. We first consider the trader properties needed in an isolated subsystem, and then augment the model to cover multiple autonomous systems. We close with some words about our trading software, which is not restricted to the trading services only.

2 Trading service

The trading function supports location transparency of services, even if the server objects are located in separate, autonomous systems. The system boundaries can correspond to administrative, technical or political areas. Traders provide a network wide selection among interchangeable service providers. A trader selects a server object for a given task by considering the dynamic status of the service providers, the service requesters and the underlying system infrastructure. Traders differ from the name and directory servers by offering a nondeterministic translation from an attribute based description of a required behavior to a location of an object interface. Many dynamically chancing factors affect the decisions of a trader, and it can produce different results depending on requester or time.

A *trading community* consists of a group of objects, from which one plays the trader role, and other are in the importer and the exporter roles. Exporters announce to the trader the abilities of servers by *exporting* service offers. The *service offer* includes service properties, that describe an abstract service, and server properties, that describe an individual service provider. The server properties can be static or dynamic: static properties describe the abilities of an object, dynamic properties describe the status of a server. The exporter can also inform the trader that the server is no more available by *withdrawing* an offer. Importers may send *service requests* to the trader to search a group of service offers, constraining their suitability by giving some matching rules expressed using the properties (*importing*). The matching rules are restrictions over both service and server property values. Importers may be themselves the objects that wish to act as clients towards the selected server(s), or work in behalf of a client object. Trading community is illustrated in Figure 2. The roles of clients and importers are shown as separate, although the same object may act in both roles. The combinations of the roles are indicated by the dotted lines.

An example trading community utilizes service types such as printers, documents and editors. The printer objects are characterized by properties like their queue length, the paper sizes available, and their location. The documents are characterized by their author, the time of last update, the storage format, and the storage identifier (file or directory name, disk name, computer name, etc.). The editor objects have properties like the required terminal type, and the offered document formats for loading and storing. In a more equipment related environment, the proper service types include bridges, cables, routers, and computers. The administration tasks utilize the exporting and the withdrawing operations of traders to control the routing of data flows in the network.

The current ODP trading function document [6] expresses, that the service offers include information that may lead to a contract between the service providing and the service using objects. We suggest that the service offers and the import requests are modeled as contract templates. An ODP contract is a description of the object behavior during an interaction between objects [4]. Thus, for an interaction, each participating object must possess a corresponding contract. A contract template is a collection of details needed in the creation of a contract [4] (<X>-template). Both the service offers and the import requests specify predicates on which a contract can be created. The matching pair of contract templates contains enough information to actually create the contracts by combining the offer and the request. The task of the trader is the matching of contract templates, i.e., negotiation of the contracts.

A trader can perform this negotiation in any environment. The information contents and the contract semantics are not defined in traders themselves, but in a separate type repository. Especially, in the ODP infrastructure, the trading function is part of the binding activity between objects (Figure 1). The binding between two or more objects is created so that each side gives the trader enough information for negotiation of possible bindings. A client has to initiate the establishing behavior of the binding between the client and a server. When the trader has found a suitable set of contract templates to form a contract, it informs the client. The client then creates the contract and informs the selected server that it is required to create a corresponding contract and to follow the behavior stated in that contract. The binding is now established.

In the DRYAD experiment, we have decided to include the following information into the contract templates (i.e. into the service offers and the service requests): (i) a list of logical operations available at the offered interface, (ii) a list of operation representations available at the offered interface, (iii) information about the object offering the interface, such as its location, the technical details of invocation mechanism, a list of other interfaces on the same object, and dynamic status information.

3 A single trading domain

In this chapter, we consider the structure of objects participating a trading community. The group of objects that play the roles of traders, importers, clients, exporters, and servers is called a trading community. A system controlled by a single trader object is called a *trading domain*. In this chapter, the domain and the community consist of the same objects, but later we let the community to span several domains. The trading community has already been introduced in Figure 2. A computational viewpoint description of the trading community and its object templates is shown in Figure 3.

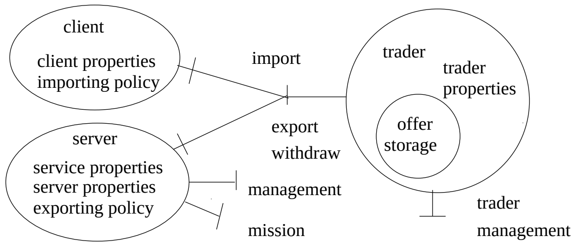

Figure 3: Trading community object templates.

```
- trader security policy
- trader acceptance policy, consisting of
    - service offer acceptance policy
    - import request acceptance policy
- type policy
- arbitration policy
- export action policy, including
    - storage policy
- import action policy, consisting of
    - local search policy
    - resource consumption policy
    - domain boundary crossing policy
    - global search policy
```

Figure 4: Summary of trader properties.

A trader template (model) consists of four components: (i) templates for operations import, export and withdraw, (ii) storage for offers, (iii) set of trader policies guiding the behavior of the trader, arranged as trader properties, and (iv) templates for trader management operations.

Some of the trader properties shown in Figure 4 are private to the trader, i.e. the trader does not expose them to other objects, but uses them as a behavioral guideline. Some other properties are public and can be revealed to objects requesting trader properties.

The trader community also includes clients, importers, exporters, and servers, as roles. Therefore, each object template includes the following (possibly empty) components: (i) exporting policy, (ii) importing policy, (iii) security policy, (iv) public properties, (v) private properties, (vi) mission specific operations, and (v) management operations.

In the ODP framework, all objects have a security policy. It restricts all communication initiatives from other objects. Within the trading domain, it is important that all importers and all exporters utilize similar knowledge about service types as the trader does. Therefore, each trader is required to publish the used type repository as type policy value.

The collective behavior of the community at the import, export, or withdraw operation is regulated by the policies of each participating object. Also the importers and the exporters can thus regulate the result of the trader's contract negotiation process. Therefore, we introduce the trading community policies by describing the trader behavior at each operation.

The trader's behavior at the export operation is regulated by service offer acceptance policy and storage policy. Besides the object level security check, the trader has internal considerations towards its importers and exporters. These internal rules are collectively called trader acceptance policy, and they guide the trader to give differing replies to requesters depending on their identity. The service offer acceptance policy prevents untrusted service providers joining the trading community. To implement its purpose, the trader must have a storage for offers. The storage organization is an administrative decision. It can be decided at the time of trader software implementation. The logical structure of the storage is expressed as a trader property called the storage policy. Storage policy can stipulate arranging offers into groups, for example, by service type or other predicate, or require hierarchical storage with contexts representing organizational boundaries.

The exporters must also have an operation on the trader to withdraw their offers. The decision to withdraw is done by the exporter alone. It is possible, that some clients in the community know and hold the service offer, when the exporter decides to withdraw it. The consequences of offers becoming stale can be reduced by the service offer acceptance policy and the importing policy. The service offers might be required to carry a property value implying the probable time to live, and the service request might carry a predicate requiring an adequately long availability time for the selected offer. If it is desirable that the trader knows which clients hold copies of a service offer, other agents within the trading community (importers, exporters, or new agents) can provide the necessary information. These agents can modify the properties of a service offer or provide dynamic values for the offer properties.

The exporters are able to restrict the set of objects allowed to import their offers. The exporter has an exporting policy associated with it. This policy can be stored in the service offer as a set of predicates considered by the trader when selecting suitable offers. The exporter may, for example, restrict the access to a service for a particular organization only, although the federated trading community may span several organizations.

The import operations may vary in the extent of information returned by the trader. The importer may retrieve only the identifiers needed for binding, or also the property values of the offers. The execution of the import operation requires that the trader import request acceptance policy allows the trader to attend the importer's request. If allowed, the trader searches from its storage a set of service offers that can be joined with the service request to form a contract. The trader's matching process is guided by (i) the importer's importing policy, and the predicates in the service request, and (ii) the import action policy of the trader. The set of acceptable service offers must be further checked, so that also the service exporter can accept the service import to take place. The local search policy regulates the search order within the local offer storage and may, for optimization reasons, restrict the search to certain subcontexts of the storage. The local search policy can be decided by the administrator of the trader, within the selection possibilities that are made available by the trader software implementation. The search is also restricted by the maximum resource consumption allowed for an individual import request. The local search policy and the resource consumption policy are the main components of the import action policy.

In all operations, the arbitration policy settles contradictions between simultaneously affecting policy rules. Depending on the arbitration policy, the trader can in effect get different roles. The trader can become a dispatcher, a trustee, a consultant or a coordinator [13]. The trader can assume different role in each import action. Arbitration policy can also include components that show how selection is done if the service request predicates do not produce a selection narrow enough.

4 Multiple trading domains

As the global network is composed of several autonomous, heterogeneous systems, an implementation of the trading functionality can not be centralized. Each autonomous subsystem should have a trader controlling its services. The administrator of any trader may independently select service types, that are known in the trading domain, and policies, that affect the storing and searching of service offers, the security and the participation in cooperation between domains.

To achieve the location transparency of services throughout the combined network, traders have to cooperate. The engineering of the trading service may be different at the separate autonomous systems and the technology with which the trader has been implemented may differ. Also the requirements set by the trading domains in regard with the trading service quality, e.g., cost, speed and accuracy of the relayed offers, may vary. The policies regulating the behavior of the trader may also differ from a subsystem to another. The cooperation of traders supports federation transparency, i.e. hides from the trader users the differences of the cooperating domains. This property gives tools to the ODP infrastructure to provide the distribution transparency of all services.

The ODP reference model offers interceptors to mediate between objects using different communication mechanisms or different service type specifications. Figure 5 illustrates that the trader objects of otherwise isolated trading domains can communicate when they are bound together via communication domain interceptors (light-weight protocol translators). The interceptors translate any data representation differences in the requests and the replies. These interceptors are necessary to introduce into the systems before the traders can communicate. When an importer requests for a server of a certain service type, the trader has to make sure that the remote trader

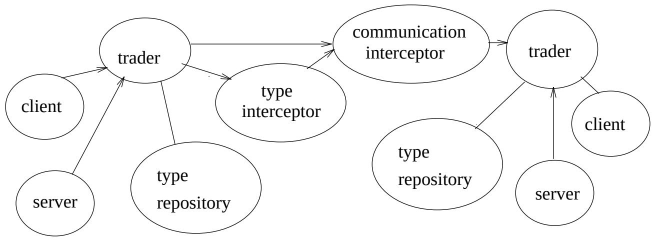

Figure 5: Interworking of traders.

understands the request. Each trading domain has an internal agreement of available service types and communication methods. The traders can negotiate about the known service types using the help of the local type repository functions. As a result of this negotiation, a type interceptor can be created and introduced into the communication between the traders.

5 Trading communities spanning across multiple trading domains

Cooperation between trading domains (i.e., federation) causes only one change in the trader behavior. The import operations may now expand several trading domains, and therefore, the trader must obtain information about other trading servers.

If a trader works alone when serving an import request, it needs only to search its own storage of service offers. If it has decided to use federation, it relays the import request to other traders. The relaying trader acts in an importer role in regard to the other traders, on behalf of its own importer. When the import operations terminate, all the resulting sets of offers are passed to the trader in importer role. The trader may temporarily store the offers for further use, but it is not obliged to do so. The trader may request several other traders either in parallel or serially. When these requests have been served, the trader collects all service offers, possibly makes some selection among the offers, and then sends the information to the importer as the response to the original request. The decision what other traders to use is based on the global search policy used by the trader. It includes information about finding suitable cooperating traders. The domain boundary crossing policy contains information about types known in the remote domains and possibly also hints for deciding what requests are worth passing to each trader.

Some difficulties arise, because the trading domains are autonomous. They are allowed to independently decide which importers are served. However, they are not aware of the identities and naming schemes of object in the remote domain. Furthermore, the traders have to compare service requests and service offers originating from separate domains, which may independently decide what service types are supported. The first problem can be solved by letting the trader to represent the whole domain as an importer in a federation situation. The administrators of remote traders have to allow the local trader to act as an importer, by the normal trader security policy. The local trader is then responsible of checking identities and trustworthiness of objects within its own domain. The second problem is solved by utilizing a type repository for comparisons of type specifications. However, it is not reasonable to compare the whole type space every time a federated import is attempted. Neither it is reasonable to compare whole type spaces with every possible federation companion in the network beforehand, especially if the network is large or changing over time.

Some cooperation establishment mechanisms, that are suitable for small or relatively static configurations of traders, have been introduced. The federation model of [1], [2], and issue list of [3] uses two protocols: one between the importer and its local trader, and another between two traders. The protocol between two traders includes active exporting of trading service offers to all potential cooperation parties, negotiation of contracts and storage of the resulting federation export and import contracts into the potentially interacting traders. Thus, even the traders that probably do not use the possibility of cooperation establish these federation contracts. The traders store knowledge about remote traders as as a link domain consisting of trading service contracts. This is in analogy with the service offer domain that consists of service contract templates. At the import operations, the local trader acts on behalf of the importer and initiates an import operation at the cooperating traders. The cooperating trader offers special operations for importing traders that correspond to the import operations for local importers. The duplication of import operations is used because service types must be intercepted when the import activity crosses a subsystem boundary.

In this scheme, a trader stores knowledge about all cooperating traders and compares beforehand the type spaces in order to decide what service types are compatible, and what interceptors are needed to create mappings for technical differences. In order to obtain this knowledge, the trader must behave in the following way: Let us consider federation among N traders, each having M known service types in common. Each liaison between two traders requires storing of a tuple { target trader identifier, service type available, interception knowledge, restrictions for importing }. Each liaison requires a broadcast in the global network to reach possible traders, followed by bilateral negotiations to form a contract about all M service types and interceptors needed at the federated import operations. This means N broadcasts and N x N x M negotiations in total. After negotiations, each trader has a storage of N x M tuples, regardless whether the liaisons are going to be used or not. The negotiations must be repeated from time to time to refresh the knowledge.

We suggest a simplified mechanism, where partial comparisons of service type spaces is done by demand. When a trader wishes to import offers of a certain service type, it requests possible cooperators from a repository of trading servers. (Traders are by nature servers themselves. Traders may import service offers also for their own needs, not only for the needs of their importers. It is a security policy decision, whether the trader is allowed to use trading service offers of new and possibly untrustworthy systems imported from the trusted neighbor trader.) It also requests descriptions of the service types those traders support. From these replies the trader is able to decide, which other traders offer compatible service types and what interceptors are needed. It may store this information for further use.

To make the requests for service types possible, the trader properties must include the type policy. Included in the import action policy is a domain boundary crossing policy. This policy consists of a reference list to known cooperators, each with an incremental list of intercepted service types. In other words, the trader is aware of the service types available locally and also aware, how to intercept service type information when importing from another trader. The trader collects the interception information incrementally.

Let us again consider the cooperation among N traders. Each trader now imports offers from K other traders. When an importer requests a service, the trader first imports a set of trader offers. Next, the trader negotiates directly with the selected K traders and stores a maximum of K x M tuples. No broadcasts are needed, and a maximum of N x K x M negotiations may take place on demand scattered over a longer time period. In a typical case, N is large, but K and M are small.

The key issue in cooperation of traders is the dynamicity of links connecting cooperating traders. A link has to reflect the availability and the reachability of the remote trader, without costly polling protocols. This is achieved by not storing the remote trader addresses in the links directly, but by importing from the trader offer space the trading service offer details as needed. This makes the verification of the link easy. It is convenient for the trader to automatically find new traders from the network. Also, the trader administrators are allowed to migrate their traders without negotiating with other trader administrators.

The traders can themselves spread the locations and properties of trading servers. The traders can assume the exporter role with regard of a few neighbor traders. However, there is a need for some inherent knowledge to start with. The administrator may present some neighboring traders to the local trader, and the trader itself is exported to some other traders. As traders from time to time import trading service offers from each other, they gradually learn to know newly created traders and can start to cooperate with them. Each trader has a partial view of the global set of traders, and no central repository exists. Thus scaling problems are avoided. The method resembles the probabilistic sparse diffusion multicast presented in [11]. The spreading of a trading service offer is restricted by the same mechanisms as normal services. This exporting system contains no active negotiation of federation contracts.

By cooperation, an importer may now receive from the local trader a set of service offers originating from separate autonomous systems. As the trader behavior in the import operation has now been changed, we must consider also the corresponding predicates that the importer may set for the operation. The importer may guide the trader in federation by either allowing or forbidding offers originating from some cooperating traders, or by allowing a certain amount of hops from trader to trader. Some restrictions, such as cost and time, also apply when the trader passes the request to other traders. The trader must then split the maximum allowed resource consumption between traders involved. From an exporter's point of view, an exporting policy may allow importers from federating domains, or deny all foreign attempts.

6 Trading software

The ODP trading function specifies only the requirements of pure trading service and the trading service management actions. Software that supports trading must also include a type repository and its management, and a policy repository and its management. These six functions trust on technology giving the storage function and the communication function services. The DRYAD software implements a composition of the trading, the type repository and the policy repository functions. Although, the policy repository is not recognized by the ODP model, we have wished to keep it as a separate function, because we plan to utilize the same policies shared between trading and service invocation functions.

The DRYAD project has produced an experimental trader implementation in a general UNIX environment using a light-weight database management system [7]. The implementation supports dynamic service and server property values, caching of these values, and federation issues [12]. The software also includes a service browser and a management tool for traders and service types [8].

Let us finally consider some details of our trading software, in order to show, how policies, cooperation of traders and usage of trading services can be arranged.

Trader policies fall into two groups: some of them are arranged already in the implementation, others can be managed via a graphical management tool. The storage policy of DRYAD trader is fixed by implementation: the storage is structured by service types, and no backups are kept, except on the trading service offers. Also the security policy is fixed. It trusts on the UNIX security services. Normally we allow only the system managers to modify policies. Via the management tool the trader administrator is able to program the acceptance policies and adjust the arbitration policy. Policies are programmed as a set of rules that the trader is able to interpret. For example, we could state that CS department owned objects may export services, but all trading service offers are accepted by first negotiating with the administrator. We could also write an import request acceptance policy stating that only CS department users may import all services, but named individuals from other organizations may import document services. The arbitration policy is adjustable at user, importer, service type and server basis. Each rule, that is included into the policies, has a priority that shows which other rules it may override. Users may have restrictions on setting their policy priorities. In a general case we allow the trader policies to override importing policies.

Also the type repository is created with a graphical management tool. Service types that we already have used include expert mailboxes, automatic news clips delivery, printer selection and automatic and distributed demonstration program library. We have also plans to trade CPU's, based on load sharing.

In order to facilitate cooperation with other trading domains, the administrator must export some trading service offers to the local trader. This initial exporting is required only once, and it can be done by a simple program contained in our software package. The management tool can be used to introduce the required type interceptors, too.

We have created a library package to support programming of importers, but the clients can also be supported so that old applications can be integrated easily. Exporting can be included in system bootstrapping processes.

7 Conclusion

In this paper, we have described trading service that offers information about available service providers. The trading service can be used to achieve interoperability between autonomous systems. Each subsystem, i.e., domain, includes a trader that controls the services on that domain. A domain can be formed on administrative, technical or political basis. We have described a simple model of trading function that allows cooperation between autonomous trader objects. This extends the potential area of service providers used. Basic goals of the model have been to minimize the amount of statically stored cooperation information in each trader, and to use same policy regulated import operations for local and cooperated trading.

The trading function is currently under standardization process. We think, that the constellation of cooperating traders is a matter of standard usage, not a matter of standard itself. Interworking traders are under international co-research and prototyping. This research aims to produce a set of practical trader networks that mediate the mostly needed service types.

Acknowledgments

The DRYAD project has received financial support from Finnish Technology Development Center TEKES and four Finnish industrial parters during a two year study period in 1992-1994.

References

[1] Bearman, M., Raymond, K., Federating Traders: An ODP Adventure. International IFIP Workshop on Open Distributed Processing. Berlin, 1991.

[2] ISO/IEC JTC1/SC21/WG7 N807. Working Document on Topic 9.1 - ODP Trader. June 1993.

[3] ISO/IEC JTC1/SC21/WG7 N897. Information technology - Open Distributed Processing - ODP Trading Function. WG 7 working document. November 1993.

[4] Information Technology - Open Systems Interconnection, Data Management and Open Distributed Processing - Basic Reference Model of Open Distributed Processing - Part 2: Descriptive Model. Committee Draft. November 1993.

[5] Information Technology - Open Systems Interconnection, Data Management and Open Distributed Processing - Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model. Committee Draft. November 1993.

[6] Information Technology - Open Systems Interconnection, Data Management and Open Distributed Processing - Draft ODP Trading Function ISO/IEC 13235: 1994. Committee Draft under consideration. July 1994.

[7] Kutvonen, P., Kutvonen, L., Debbie, A database management system. University of Helsinki, Department of Computer Science. Report of DRYAD project. August 1993.

[8] Kutvonen, L., Supporting transition to open, heterogeneous network environments. To appear in TINA'95. Melbourne, February 1995.

[9] Kutvonen, L., An infrastructure for distributed service invocations in a heterogeneous network. Department of Computer Science, University of Helsinki. Report of DRYAD project. July 1994.

[10] Raymond, K. A., Reference Model of Open Distributed Processing: A Tutorial. International IFIP Conference on Open Distributed Processing. Berlin, 1993.

[11] Schwartz, M., F., Resource Discovery in the Global Internet. Department of Computer Science, University of Colorado, Boulder. CU-CS-555-91. November 1991.

[12] Venetjoki, T., Cooperation of traders in DRYAD. Department of Computer Science, University of Helsinki. MSc Theses. April 1994. In Finnish.

[13] Wolisz, A., Tschammer, V., Dynamic Binding of Service Users and Providers in an Open Services Environment. Computer Networks, 1991.