# Overview of the DRYAD trading system implementation

*Lea Kutvonen*
*Department of Computer Science, University of Helsinki*
*P.O. Box 26, FIN-00014 University of Helsinki, FINLAND*
*Lea.Kutvonen@cs.Helsinki.FI*

**Abstract**

The Open Distributed Processing Reference Model (ODP-RM) of ISO and ITU defines a set of basic services which every open system has to support and use. One of such services is the trading function. The purpose of the trading function is to help objects, for example application program components, to discover target objects in a dynamic network environment. In this paper, we introduce trading as means to establish contracts between objects. Contract establishment facilitates distribution transparent access of services over autonomously administered subsystems. We also discuss an experimental prototype of an ODP like trading system. The platform for the DRYAD trader is a UNIX environment supplemented with a special-purpose database management system.

**Keywords**

Open Distributed Processing, trading, contract

## 1 INTRODUCTION

Interworking between autonomous systems relies on support from an information service that can provide dynamic knowledge about available service providers within the network. The trading service (ISO, 1994), as described by the Open Distributed Processing Reference Model (ODP-RM) (ISO, 1995), offers such a service.

In the DRYAD project we have studied the ODP infrastructure model and implemented prototypes of some ODP services. Figure 1 illustrates the components addressed: service invocation, binding, trading, and type and policy repositories. The communication layer is realized by the operating systems facilities. We consider these as the fundamental functions needed for distribution transparent service interoperation in heterogeneous environment.

The service invocations cover the announcements and the interrogations as they are described in the ODP-RM (Kutvonen, Feb. 1995). Service invocation relies on implicit binding services. Each object in the system (e.g., a component object of an application system) has one or more server role interfaces offering operations for other objects to invoke. Also, each object has client role interfaces that represent views to the used services. The infrastructure services support these views. The type repository functions verify

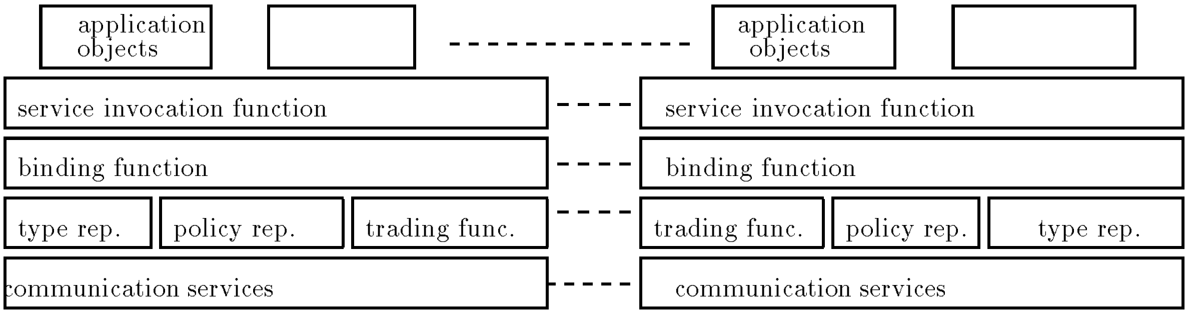


**Figure 1** A schematic view of an ODP like infrastructure in the DRYAD project.

which client and server role interfaces are semantically conformant. The trading function selects a suitable service provider for each service request. The objective is to increase the availability of an abstract service by providing a choice from a set of real servers in the network. The binding function uses the distributed trading service and, based on that information, creates channels between objects. When the objects are bound together, operation invocations may occur.

We have started to populate the ODP infrastructure by first prototyping the layer with the trading function. The trading software includes the DRYAD trader object and the DRYAD trader user library, the Nereid graphical management tool, and the Artemis graphical browser (Björklund, 1995).

In this paper, we define the trading function in terms of contract introduction and negotiation. Contracts are essential for object binding and service invocation in distributed environments. We also explain how the DRYAD trading system implementation reflects the current trading function committee draft (ISO, 1994). We show the contents and structure of repositories required for the trading system. The language used at the trading interface is described, because this issue is not defined by the current trading function committee draft. In order to show how the trader components work together, we describe the behavior of the trading operations.

## 2 TRADING SERVICE

The trading service is a general tool for flexible object discovery in a dynamic network environment. The objects 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 differs from an advanced directory service by allowing an attribute-based search where the attributes can be evaluated at search time.

The basic trading operations are import, export and withdraw. The export and withdraw operations are used for adding and deleting service offers to and from the trader's offer repository. These operations can be invoked by the offered server object itself or another object working on behalf of it. Hence, there is a generic 'exporter role'. The import operation is needed for obtaining information from that repository. Again, the 'importer role' can be assumed by the client itself or someone acting on its behalf. A service offer includes the identity of the server, and description of its properties.

The import request includes the importer's criteria for selecting a set of offers. The criteria consists of matching criteria, i.e., strict requirements for offers to be selected, and preference criteria for choosing the preferred ones from a set of offers fulfilling the matching criteria. The importer may also restrict the search area, or the cost of the search.

The trading service is not limited to any specific object technology or any specific application area. It is a general mechanism to match client requirements with server abilities: the trading function matches arbitrary data values, and the type repository supports this process by supplying application area specific semantics and structure for the values.

## 3 SEMANTICS OF THE TRADING SERVICE

The foundations of the overall DRYAD architecture for invoking services are defined in the ODP-RM concepts for object, interaction, distribution transparency, contract, liaison, domain, and autonomy of subsystems (ISO, 1995).

In our approach, a distribution transparent service is required to be access transparent and location transparent. Access transparency masks differences in data representation between objects and in invocation mechanisms between subsystems; location transparency masks the locations of object interfaces during interactions. Although distribution transparency is supported by the infrastructure, the objects must support some features of distribution when they interact.

Every object that accesses services includes contracts. A contract is part of the client role interface. The contract describes the behavior of the client object itself during the execution of operations at the interface. It also describes what the infrastructure environment is required to do. Correspondingly, a server object includes a contract as part of its server role interface.

The contract consists of two parts. The first part, **the environment contract**, defines rules for communication with the infrastructure components. The second part, **the service access contract**, defines the rules for client-server level communication. The infrastructure needs a corresponding set of contracts with service providers to be able to behave in accordance to the client's contract. In the environment contract the client determines a set of communication service properties. For example, quality of service (QoS) attributes, failure scenarios, and aspects of communication synchrony need to be defined in the environment contracts. In the service access contract, the client determines a set of properties it requires from the service provider. For example, application level protocols need to be defined in the service access contract.

The contracts have three distinct modes: **a potential contract, a (real) contract, and a liaison**. When the client is prepared to request a service or a server is prepared to provide a service, they include potential contracts. A potential contract describes the preconditions of the actions that the object is prepared to take, and also, the object behavior when the preconditions are met. A (real) contract is created when a client has requested a service and the infrastructure has found a suitable set of potential (server) contracts. A potential contract only names either the server or the client, but in a (real) contract, both sides are known. A liaison is formulated by establishing the (real) contract at each involved object. A liaison is also called a common contractual context, and it means that each involved object knows that also all the other objects behave according to the same (real) contract.

If the server's and the client's environment contracts are not compatible with each other, they might still be negotiable. When external help is needed to mediate between environment contracts, interceptors are used. An interceptor is an object that is able, for example, to translate representation formats (ISO, 1995).

Trading supports the service invocation process by implementing the negotiation of potential service access contracts. A potential service access contract can be exported as a service offer to the trader system. The potential contract includes conditions on which the server will form a contract with a client, together with a description of the server's behavior when the contract is in use. Also the importer passes a potential contract to the trader, as a service request. The potential contract includes conditions restricting the set of servers with which the client is willing to establish a contract. The trader matches these potential contracts, forms a contract, and returns it to the importer. The importer can then take further steps in order to initiate contract establishment. The resulting liaison can further be realized by a binding between objects. The binding is required for service invocations to succeed.

Because we use trading service as part of the liaison establishment process between clients and servers, we do not allow long lived imports from the trader. If the importer needs to hold an offer for use in further service invocations, the exporters and the importers should have special policies and protocols for these cases. The long lived imports are not a service supported by the trader, but instead a property of the formed contract.

# 4 DISTRIBUTION ASPECTS

The trading function is a global service. It may span over several organizations and trade for very different styles of services (operating system services, applications, information objects). Therefore, the work needs to be divided for several traders. For such a distribution, the concept of domain is used: a trading domain is a set of objects, importers and exporters, in liaison with one trader object. The trader objects may cooperate without loosing control over their own domains by assuming importer role in each others' domains.

The trading functionality can be viewed at two levels of abstraction. On the higher level, the global service is seen as one trader object. On the lower level, we recognize multiple administrative domains, each having their own trader objects. These trader objects federate, i.e., interwork, to fulfill the task of the global trader object. The partitioning of the global system into domains offers a set of independent access points to the trading service. Within each trading domain, there is one specific trader object and a set of exporters and importers depending on that specific trader. That trader serves as an access point to the global trading service for the local exporters and importers.

# 5 ROLE OF THE TYPE REPOSITORY FUNCTION

It is important that the exporters, the importers and the trader have a common understanding of the service requests. The purpose of the type repository function is to support knowledge about service types, their semantics and their technical representation. The repository also allows dynamic modification of the available service types in the system.

A service type is described by a tuple of an interface signature type, related behavior and a set of service properties. An interface signature type can be either an operational or a stream interface. The operational interfaces can be described with interface description languages (IDLs) by naming each operation and their parameters. For behavior descriptions no generic methods exist.

Because the tools for generic service type descriptions are lacking, the trading function is required to match offers based on interface signatures, and optionally also based on service type names.

The type repository is partitioned into domains, for similar reasons as the trading function. Each trading domain is contained in a single type management domain, but there are no limitations for having several trading domains under the same type management domain. If two traders, that are included in different type management domains, interwork, they may need type translation services. Because the traders are used to support distribution transparency, they must be aware of type management domain boundaries.

# 6 CONTROLLING THE TRADING BEHAVIOR

The trading function committee draft (ISO, 1994) defines the trading activities only in terms of external behavior. Different realizations are allowed to have a slightly different internal behavior. Therefore, an importer will experience a nondeterministic behavior in a trading network. For example, 'the best fitted' server is chosen differently in different domains.

However, each trader object behaves deterministically. For federation establishment information about characteristic behavior of each trading domain need to be available. The characteristic features are represented as policies. Many of the policies are traditionally considered as configuration information, like internal storage structure and access lists.

A policy framework is used for the management and description of the trading behaviors. Policies are simple rules or flags that can be interpreted by all objects. The language for the policy expressions may vary between administrative domains. The framework has three dimensions for the classification of policies: (i) policies of importers, exporters and traders, (ii) policies involved in importing or exporting, and (iii) policies for local operation, security, and federation. The framework does not force implementations to have policies adjustable by parameterization, but it allows the trader's policies to be announced through the trading mechanism itself. (For a more detailed discussion, see (Kutvonen, Apr. 1995).)

Importing or exporting offers pass to the trader also information about the policies under which the importers and exporters themselves work. In the trader's selection process, the policy rules given by importers, exporters and the trader itself, are merged together. The import operation parameters carry the policy rules stated by the importer. The exporter may store policy rules into the service offer. The trader administrator can set the trader policy rules.

The second classification introduces import and export action policies. Most importantly, an import action policy defines which import requests are attended. In addition, it states which set of the matching and preference criteria are used to select the suitable offers and how much resources an individual import operation is allowed to consume. It also tells which remote traders should be searched. An export action policy guides the trader's behavior during export operation: the trader can accept or reject an exported service offer; for storage, the trader has to define an internal storage structure; the trader is also allowed to withdraw offers due to its internal technical reasons.

The third classification adds security related and federation related policies. In addition to the security function of the ODP infrastructure, the trader can further restrict the group of importers and exporters it attends. Federation policies affect only the import actions by guiding the search to other trading domains.

In case that the objects involved define contradicting policy rules, there must be an arbitration policy. The arbitration policy defines, which policy rules override others.

When building an object interworking environment, the existence of a policy repository becomes very important. All essential policy decisions in such an environment can be done at the trading function level.

# 7 THE DRYAD SOFTWARE STRUCTURE

## 7.1 General composition

The key components of the DRYAD trading software are a trader object that offers service and management interfaces, a library interface for C programmers, and graphical management tools for the administrators. The software is an operational prototype. The management tools are independent of the potential application areas and the potential service type descriptions. Therefore, the administration of DRYAD based trading applications requires expertise. The trader object and the trading library components are written in C and run in SPARC/Solaris environment (ONC, Open Network Computing) and on Linux (Welsh and Kaufman, 1995). The graphical user interfaces run in SPARC/Solaris environment. For storage, the trading system uses a portable, special-purpose database engine, called Debbie (Kutvonen and Kutvonen, 1996).

The DRYAD trader object (Figure 2) consists of

- a trader repository object, that has two parts: the offer space contains information about offered services, and the trader address storage locates traders in the network,
- a type repository object, that includes the definitions of service types,
- a policy repository object, that includes trading policy components,
- a template repository object, and
- a concierge object.

The repositories are aggregated into one object. The object's access interface is controlled by a concierge, and the repositories rely on Debbie database system services. Initially only one active object, the concierge, is running together with the Debbie system. The concierge watches the incoming communication ports (imports, exports, administration) and requests the Debbie engine to create a new active object to handle each incoming message. The active objects are instantiated by letting the Debbie system to execute methods from object templates that are stored in the template repository.

Each repository object has one or more active processing objects working on them. The trader repository has agents for handling import requests and offer handlers for handling export and withdraw requests. Each import is processed by a private agent instance. The type repository has a type manager to respond queries.

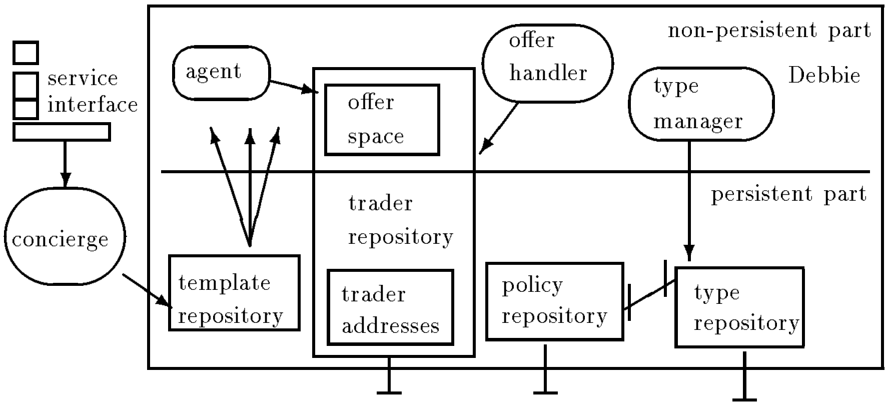


**Figure 2** The DRYAD trading system components.

The repository objects themselves are passive. The Debbie database can store both persistent and non-persistent information. The offer space of the trader repository is by a policy decision stored non-persistently. The other repositories need to be persistent, because they are needed at the trader startup.

The repository structures are introduced in the following chapters. As the type and policy repository components coexist in the same database, also an allocation scheme is shown: the implementation objects contain both a type object and a set of policy objects.

## 7.2 Type repository

The type repository is used to store information about service types and meta-information about server properties. New service types can be added while the trading system is running.

The type repository includes a type description object for each service type. The object is composed of the type name, a verbal description of the service type semantics, information about the service type's visibility to other trading domains, a set of attribute objects, and a set of interceptor objects.

The attribute objects in turn consist of the attribute name, verbal description, data value type, unit of value, and default virtual value of the attribute. The default virtual value defines the evaluation method of the actual attribute value. Technically the object also includes caching time for the evaluated values. Examples of attribute specifications are

- name = InterfaceSignature, data type = XDR description (corresponds to IDL descriptions), unit = NIL (not applicable with this data type), default value = NIL,
- name = Location, data type = set of addresses, unit = NIL, default value = NIL, and
- name = QueueLength, data type = Integer, unit = Bytes, default value = 'lpq -P$ID'.

Each type interceptor object defines the relationship between a locally known type and a type known in the remote domain. An interceptor object consists of a local name for the remote type management domain, a name for the wanted service type within the remote type management domain, and a set of rules for generating local attributes from the values retrieved from the remote trader. The transformation rules are constructed from arithmetic and string manipulation operations over the attribute values. Some functions are available for accessing the attribute type information from the remote type domain: data type, upper and lower boundaries for potential values, and unit of the attribute value. The transformation rules can also include conditional statements and comparisons.

The type repository of the DRYAD trading system is built on the assumption that service types are matched by names, and interpreted as abstract service types which can have several alternative interface types. Therefore, the importer needs to request for offers that have a certain interface signature type available. This scenario is naturally relevant only in the service invocation system. Subtyping of service types or interface signature types is not supported.

## 7.3 Policy repository

The trader policies used in the DRYAD software are defined separately for each service type. The policies are

- import action policy, that is composed of import acceptance policy, federation policy, selection policy, and resource consumption policy, and
- export action policy that includes only export acceptance policy.

The acceptance policies express potential importing or exporting objects per service types. Each request is tested against these expressions. We represent the acceptance policy as regular expressions on addresses. An address consists of a host name, a port number, a protocol name, a file name, and a process number.

The federation policy defines which other traders need to be requested for import and in which order. We represent the federation policy as a partially ordered list of trader names.

The selection policy includes system matching and system preference criteria for selecting offers. The criteria are represented with the same language constructs as the importer's selection criteria used as parameters of the import operation (see Figure 3). By applying the selection policy the trading system can be configured to have different roles over different service types. Six different roles of trader have been presented: a match maker, a dispatcher, a trustee, a consultant, a coordinator (Wolisz and Tschammer, 1991), and a secretary. The match maker works on compulsory, static data, while the others can use also dynamic values. The match maker is the simplest trader system (e.g., directory servers can be used for this purpose). The dispatcher and the trustee both do the selection on one selection rule source. The dispatcher uses only its internal rules, while the trustee depends only on the rules of the client. The consultant and the coordinator use both rule sources, in different order. The consultant lets the client select from the group of servers conforming to the trader's policy rules. The coordinator has some internal selection rules for the servers accepted by the client. In addition to these, we have defined a secretary role. A secretary can play different roles in simultaneous import actions.

In the DRYAD trader implementation, many policies can be changed at run time through the management interface. Only the export action policy is fixed: the offers are stored non-persistently (i.e., a new trader instance does not adopt the offers exported into the previous one) and an offer may be deleted by the trader itself if it can not access the offered interface. If persistence of offers is required, there must be an additional active object that forces the trader to store the offers in a stable storage. Installation of such an

```
%token identifier, string, integer, floating_point_number, REQUIRE, PREFER,
  OR, AND, IN, CAT, function_identifier
restriction : clause | restriction ';' clause;
clause : disjunction | priority ':' disjunction;
priority : integer | REQUIRE | PREFER;
disjunction : conjunction | disjunction OR conjunction;
conjunction : predicate | conjunction AND predicate;
predicate : server_identifier | expression comparison_operator expression | expression IN
  '{' expression_list '}' | expression '!' IN '{' expression_list '}' | '('disjunction')';
comparison_operator : '<' | '<''=' | '=' | '!''=' | '>' | '>''=' | '≈' | '!''≈';
expression : term | expression addition_operator term;
addition_operator : '+' | '−' | CAT;
term : factor | term multiplication_operator factor;
multiplication_operator : '*' | '/';
factor : primary | function_identifier '('expression_list')' | '('expression')';
primary : attribute_identifier | string | integer | floating_point_number;
expression_list : expression | expression_list ',' expression;
attribute_identifier : identifier;
server_identifier : string;
```

**Figure 3** Yacc definition of the criteria and policy rule language.

object would mean a change in the trader's export action policy. The system structure is suitable for such a change. The arbitration policy is realized by rule priorities incorporated in all selection criteria and policy rules. The criteria priorities are the following (in descending order): (i) the system requirements, (ii) the user requirements, (iii) the user preferences, and (iv) the system preferences.

## 7.4 Combining the type and policy repositories

When a system is implemented using several ODP functions, the implementation objects are compositions of those objects defined by the function specification involved. For the trader, we aggregate one type description object and a set of policy objects. Conceptually the aggregated object is a **template for potential service access contracts**. In the following we call it as a contract template.

The contract template describes all the necessary information for the creation of service offer objects and for the import request handling. Because the offer handler only needs information about local policies and types, and the trader works either alone or in federation with other traders, there is a need to divide the contract template information into a local and a federation related partition. For federation, there is a partition for each cooperating remote trader. These components of the contract template are further divided into several Debbie database tables.

The primary part of the contract template is the table representing its interface, i.e., listing the operations the contract template object supports. The data part of the object contains a local contract template and a set of federation contract templates. The local contract template includes

- a service type description object,
- virtual values (evaluation rules for dynamic values or static values as such),
- acceptance policy as lists of allowed servers and clients,

- import action policy object, and
- the federation setting that defines whether the local trader is allowed to show the service type in question to other traders importing from it.

The federation contract templates are composed of

- the remote trader name and the service type name that should be used when importing from that remote trader, and
- reference to the interceptor object to be used for translations of attribute values between the local and the remote presentation forms.

## 7.5 Trading system administration

The trading system administration involves type, policy and trader interworking management. None of these tasks are supported by the trader object itself. We have a separate graphical interface for them, called Nereid (Björklund, 1995). Nereid interfaces directly with the Debbie database engine and allows interactive updates for the contract template components.

The trader system mechanism is controlled by the information fed in through the management tools. The administrator must 'program' the trading system by specifying the type and policy information.

## 7.6 Offer space

The offer space includes the trading systems key information: the exported service offers. Offers appear in two varieties. One form represents a local offer from the trader's own trading domain. The other form, a proxy offer, is an offer that has been imported from another trading domain, and exists only temporarily.

A service offer consists of

- an identifier (includes service type and trading contract template names),
- methods for evaluating the attribute values, a method per attribute, and
- value cache, where each value is tagged with maximum usage time.

The service offer object is created by the offer handler object as a result of an export operation. The offer is stored until it is deleted by the corresponding withdraw operation.

The proxy offer has the same structure as the real service offer. It is created by the trader agent object executing an import operation. The agent requests a remote trader to immediately return a set of offers without evaluating any missing values. The values that the remote trader was able to produce immediately are transformed to the local representation form using the interceptor knowledge. The results are stored into the proxy offers as static attribute values. If any attribute values are missing, they are replaced with a rule, that imports the value from the remote trader and transforms it in to the local form. If large amounts of dynamic attributes are used in interworking, this may cause a very slow response and a high load for all involved traders. It would be possible to combine the remaining attributes to one import request. However, the current implementation relies on the observation that most requests do not use very many attributes (Venetjoki, 1994).

# 8 OPERATIONAL BEHAVIOR

The trading service interface offers operations import, export and withdraw. In this chapter we consider only the import and export behaviors.

## 8.1 Import operation behavior

Import operation is started when the Debbie engine instantiates the agent. The agent acquires a communication port to read from and when the agent has copied the incoming message, it releases the port and returns the port control to the concierge.

The incoming message is generated by the trading library function that is available to any application program. The message structure is the following:

- service level, which defines what level of detail the importer wants to see the service offers (e.g., server identities, all attribute values, named attributes only),
- federation mode, which defines whether the importer denies, allows or forces interworking with other traders,
- federation search method, which defines whether search is done in parallel at all traders, or sequentially from trader to another until a service offer is found,
- the service type name,
- the matching criteria, the preference criteria and the importer policy rules for selecting service offers – all packed to a shared criteria field using the language described in Figure 3, and
- a list of attribute names, if the used service level requires so.

After receiving the request message, the agent checks whether this client can be accepted. If not, it exits with no response. Otherwise, it continues by retrieving and merging the importer's matching criteria, preference criteria and scoping criteria, and the trader's selection policy.

The process of retrieving rules and merging them produces a database query sequence for the Debbie engine to execute. The query consists of steps with distinct levels of priorities: requirements and a sequence of preferences. The first part of the query sequence handles all requirement level criteria and it is executed over the service offer space. If a requirement can not be satisfied by an offer, that offer can not be visible for the importer under any circumstances. Further steps can be taken with the preference criteria and executed over the intermediate result set of offers. A step is committed only when it does not create an empty result. If an offer can not satisfy a preference criteria, it may be shown to the importer, if there is an inadequate amount of services that would fulfill also this preference rule. This interpretation is also used when attribute values can not be evaluated for some reason. A requirement can not be satisfied if the value is not known, whereas a preference is not violated by a missing value. After the agent has traversed the query sequence, only those attributes that were used in the selection criteria have been evaluated. If the importer wished to have a single offer as result, then all but one random offer are deleted. Then the rest of the attributes required for the response are evaluated. Finally, the response message is sent to the importer, and the agent exits.

The reply message includes the name of the requested service type, and a set of items where each item consists of: (i) a joint name for a set of offered servers which together are able to produce the requested service (set size is currently always one), and (ii) a set of service offer descriptions which consist of server identity and/or address, a natural language description, and a set of attribute values, as [name, value, unit] triples. Especially interface locations and signature types are included in the attributes.

If the import request has a syntax error, a denial report is sent as response. Otherwise, if there occurs a minor failure during the evaluation of attribute values, the message is only tagged as 'qualified' and all the available information is returned to the importer. At the place of an attribute value, there may appear a notification of the evaluation failure and also an explanation of the type and location of failure.

### 8.2 Proclaim operation behavior

As opposed to the DRYAD architecture where the exporter behavior is described using the ODP trading function export operation, the DRYAD trader implementation uses a proclaim operation that only identifies the offered interface and its address. The reason is simple. The export operation includes a service offer evaluator interface reference by which the exporter can force the trader object to invoke whatever operation it wishes. Currently, no standard security functions are available. The DRYAD implementation does not include any security considerations except those inherited from the runtime environment. In this situation the standard export operation would cause an easily observable way to intrude into the system.

In order to avoid this vulnerability in the DRYAD implementation, we force the trader administrator to define for each service type the operations that are allowed on the service offer evaluator interface. We consider the operating system as the service offer evaluator. As the static properties of the offers are usually similar for each offer within the service type, we used the same method for the static values also.

## 9 CONCLUSION

The DRYAD infrastructure uses trading as part of the binding process required for service invocation in a distributed environment. The service offers mediated by the trading service are interpreted as potential contracts.

The concrete services offered by the DRYAD trader interface are similar to those of other trading systems. Slight differences occur because of the simultaneous development of the prototype and the trading function standard. The DRYAD software works in common UNIX environment and uses a special database engine. Other traders, such as PC-Trader (Beitz and Bearman, 1995), Melody (Burger, 1995), and TRADE (Müller-Jones, Merz and Lamersdorf, 1995) have been developed either in CORBA or DCE environment. RHODOS (Goschinski, 1993) and Y (Popescu-Zeletin, Tschammer, and Tschichholz, 1991) traders have been developed within their own distributed operating system environments. Some details of the trading services are particular to the DRYAD trader. The DRYAD trader is able to serve in different roles within the computing platform. It can have a dispatcher role in some import operations while it can play a coordinator role in another. Another interesting feature is the usage of an approximate match of data values in the criteria language.

Further work on the trading system supplements the type repository services. We also work on performance evaluation of our trading service.

## 10 ACKNOWLEDGMENTS

We would like to acknowledge all the students who have contributed to of the DRYAD prototype system. The project has received financial support from the Finnish Technology Development Center TEKES and five Finnish industrial partners in 1992-1995.

## REFERENCES

Beitz, A., Bearman, M. (1995) An ODP Trading Service for DCE. *First International IEEE Workshop on Services in Distributed and Networked Environments*, pp. 34-41.

Björklund, T. (1995) Maintenance and Integration of the DRYAD Software. Department of Computer Science, University of Helsinki. Series C, 1995.

Burger, C. (1995) Cooperation policies for traders. *The Third IFIP Conference on Open Distributed Processing*, pp. 191-201.

Goschinski, A. (1993) Supporting user autonomy and object sharing in distributed systems: The RHODOS trading service. *The First IEEE Symposium on Autonomous Decentralized Systems.*

ISO (1994) Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – ODP Trading Function. *ISO/IEC CD13235.*

ISO (1995) Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – Basic Reference Model of Open Distributed Processing – Part 3: Prescriptive Model. *ISO/IEC DIS10746-3 — ITU Rec. X.903.*

Kutvonen, L. (Feb. 1995) Supporting transition to open, heterogeneous environments. *TINA'95: Integrating Telecommunications and Distributed Computing – From Concepts to Reality*, pp. 55-66.

Kutvonen, L. (Apr. 1995) Achieving interoperability through ODP Trading function. *The 2nd IEEE Symposium on Autonomous Decentralized Systems*, pp. 63-69.

Kutvonen, P. and Kutvonen, L. (1996) Measured performance of a special-purpose database engine for distributed system software. *Computer Communications Journal, 1, 1996.*

Müller-Jones, K., Merz, M., Lamersdorf, W. (1995) The TRADEr: Integrating trading into DCE. *The 3rd IFIP Conference on Open Distributed Processing*, pp. 459-470.

Popescu-Zeletin, R., Tschammer, V., Tschichholz, M. (1991) 'Y' distributed application platform. *Computer Communication 6, 1991*, pp. 366-375.

Welsh, M. and Kaufman, L. (1995) Running Linux. O'Reilly & Associates.

Venetjoki, T. (1994) Cooperation of traders in DRYAD. Masters Thesis. Department of Computer Science, University of Helsinki. Series C, 1994. In Finnish.

Wolisz, A. and Tschammer, V. (1991) Dynamic Binding of Service Users and Providers in an Open Services Environments. *Computer Networks Conference.*