TORBA: Trading Contracts for CORBA

Raphaël Marvie, Philippe Merle, Jean-Marc Geib, Sylvain Leblanc

Laboratoire d'Informatique Fondamentale de Lille
UPRESA 8022 CNRS
Bât. M3 – UFR d'I.E.E.A.
59655 Villeneuve d'Ascq – France
{marvie,merle,geib,leblanc}@lifl.fr

Abstract

Trading is a key function in the context of distributed applications: It allows runtime discovering of available resources. In order to standardize this function, the Open Distributed Processing (ODP) and Object Management Group (OMG) have specified a trading service for CORBA objects: The CosTrading. This specification has two main drawbacks: First, this service is complex to use from applications and second, it does not offer type checking of trading requests at compilation time. Both are discussed in this paper. The main goal of our Trader Oriented Request Broker Architecture *(TORBA) is to provide a trading framework and its associated tools, which tend to offer typed trading operations that are simple to use from applications and checked at compilation time. In that, we define the concept of* Trading Contracts, *written with the* TORBA Definition Language *(TDL). Such contracts are then compiled to generate trading proxies offering simple-to-use interfaces. These interfaces completely hide the complexity of the ODP/OMG CosTrading APIs. In the meantime, TDL contracts could be dynamically used through a generic graphical console exploiting a contract repository. The example used in this paper, clearly states the advantages brought by the TDL trading contracts: type checking at compilation time, simple to use, and providing a powerful framework for CORBA object trading.*

1 Introduction

Nowadays, building, deploying, and running distributed applications rely on a set of services/functions offered by standard middleware like the *Common Object Request Broker Architecture* [19] (CORBA) of the Object Management Group (OMG), the *Distributed Component Object Model* [8] (DCOM) of Microsoft, and more recently the *Java Remote Method Invocation* [24] (RMI) of Sun Microsystems. The main functions of such middleware solutions are synchronous communication using operation invocation, asynchronous communication through message or event passing, transaction monitors, security, persistence, and resource trading. This paper proposes an innovative framework named *Trader Oriented Request Broker Architecture* (TORBA) to trade distributed objects over CORBA.

A middleware trading function tends to provide a means to discover resources available in a distributed system, in order to dynamically interconnect at runtime the various components of an application. For example, it allows a client to find back its associated server. Such a search may be based upon various criteria, like the physical location of the resource (e.g. to find the printer service of the third floor), the symbolic name of the resource (e.g. to find the BestPrint printer), or the characterization of the resource using its properties (e.g. find a color printer faster than ten pages per minute). The conceptual contribution of this paper is to define the concept of *trading contract* in order to characterize both the resource properties, and the search operations used by client applications.

The trading function has been studied both in academic projects and industrial products. Some projects have focussed on the interest of using such a function in a large scale context in order to share resources [16]. In 1993, the ANSA consortium has discussed what the trading function should be [7].

More recently, Sun Microsystems has defined a trading function, included in the *Jini* environment [1]. Based on the easiness with Java to serialize objects, this trading function allows applications to retrieve serialized objects (like network stubs or complete services). Other research works have focussed on traders federations [5], performance [6], and scalability [26]. In the context of object trading, the first technical contribution of this paper is an innovative approach that simplifies and types the use of a trading function.

In order to standardize middleware trading function, the *International Standardization Organization* (ISO) in its *Open Distributed Processing* [11] (ODP) activity and the *Object Management Group* (OMG) have defined a specification of the functional interfaces of such a function [17] using the OMG *Interface Definition Language* (OMG IDL). This specification is mainly based on the work previously performed by the ANSA consortium [7] and the DSTC [28]. It defines a set of generic APIs for applications to export and search CORBA object references in a standard and portable way, whatever the underlying implementation. Unfortunately, these APIs are quite complex to use and very technical. Moreover, using these APIs does not provide trading request type checking at compilation time, but only at runtime. Thus, the second technical contribution of this paper is to perform trading request type checking at compilation time, improving software quality and reliability.

The objective of our work is to define and to offer a typed trading environment being easy to use from CORBA applications. In that, we have defined the *trading contract* concept used to describe typed properties (object characterizations) as well as query operations to be used by applications. The *TORBA Definition Language* (TDL) is used to define these contracts. Then, it is compiled to generate trading proxies offering simple specialized interfaces to be used from client applications. The use of these interfaces is checked at compilation time, based on their types (i.e. operation synopsis). Furthermore, these proxy implementations completely hide the technical complexity of the ODP/OMG trader interfaces. In the meantime, TDL contracts could be stored in a TDL repository, like OMG IDL definitions are stored in CORBA's Interface Repository. Then, this repository could be used dynamically from a graphical console to discover available trading offers and to use defined query operations.

Section 2 of this paper presents an overview of the ODP/OMG CosTrading service. This overview focuses on trading offer typing and use of the query operation, in order to outline their drawbacks: technical complexity and lack of type checking. Section 3 discusses the *trading contract* concept, the TDL language, the proxy generation and execution process, as well as the dynamic approach. It also presents the implementation of TORBA, using a printer service as example to underline the benefits of our approach. Since TORBA use has only been performed using simple examples, section 4 presents some empirical results. Section 5 discusses the related work in middleware that are used in TORBA: the proxy concept, the structure of ORBs, and the component-oriented approach. Finally, section 6 summarizes this paper, in progress, as well as fore-coming work directions.

2 The CosTrading Service

2.1 Overview

The ODP/OMG CosTrading service is similar to a search engine for CORBA object service references. Figure 1 presents the CosTrading standard use, composed of four steps. (1) Service designers define their service offer types (see section 2.2). (2) Service providers or application servers characterize and export their service offers using properties describing the service. (3) Service users or client applications search service references using criteria describing their requirements. (4) Once references have been retrieved, clients invoke operations on the services. All these requests—definition, export, lookup, and use—are carried by CORBA.

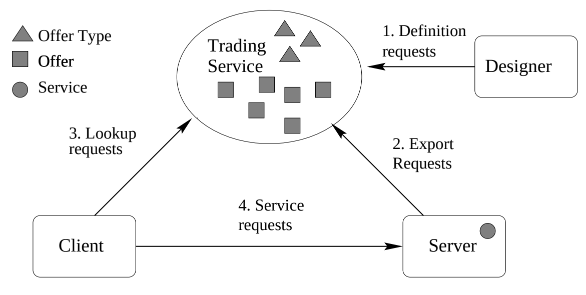

Figure 1: The ODP/OMG CosTrading Service Use.

The CosTrading service provides three main interfaces for applications. The `ServiceType-Repository` interface is used to define and manage service offer types. The `Register` interface is used to export service offers. Finally, the `Lookup` interface is used to search exported service offers. Other interfaces are also available for administration purposes, like to set the behavior of the trader and its search operation, as well as to build trader federations—offering a potentially large-scaled and unified trading service [2].

2.2 Service Offer Types

In the CosTrading, service offers are strongly typed. Any export operation is based on the use of an offer type, similarly lookup operations are performed upon a given type. Typed offers bring two main advantages. First, applications cannot export or search weird offers, but only offers defined at design time. Second, a CosTrading implementation may take advantage of types to improve performance, only searching in offers of a given type instead of in all the offers. This becomes vital when several thousand of offers have been exported. Part of the CosTrading service, the **Service Type Repository** stores the various service offer types. It also provides type checking when exporting or searching offers. In this repository, a service offer type is characterized using four elements:

- a **name**, which is a unique global identifier in a trader federation and used to define, export, and search service offers,
- some inherited **super types**, used with particular rules for redefinition which are not discussed here,
- an **OMG IDL interface** to which exported service references have to conform, and
- some **service properties** characterizing the exported service.

Each service property is characterized using:

- a **name**, which is also a unique identifier in a service type,
- an OMG IDL **type** which characterizes the type of the property values, and
- a using **mode**, which has to be set to:
  - *normal*: Giving a value to such a property is optional at creation time. If a value is given, it could be modified or removed during execution by the service provider.
  - *readonly*: It is not required to give a value to such a property, but if so the value cannot be modified.
  - *mandatory*: The service provider has to give a value to such a property at exportation time.
  - *mandatory readonly*: The provider has to give a value to the property, which cannot change during the execution.

Figure 2 presents the OMG IDL interface of a printer service. This service is used in this paper to illustrate various aspects of the trading service and our TORBA proposal.

```
interface PrinterServer {
    void print (in string filename) ;
};
```

Figure 2: OMG IDL of a Simple Printer Service.

The related service offer type is Printer, which is characterized by the four properties presented in Table 1. The `color` property specifies if a printer could print in color or only in B&W. The `cost_per_page` property contains the cost to print a sheet of paper for this printer. The number of pages per minute a printer can produce is contained in the `ppm` property. Finally, the `queue` property is the name of the printer queue.

name	type	mode
color	boolean	normal
cost_per_page	float	normal
ppm	unsigned short	normal
queue	string	normal

Table 1: Printer Service Offer Properties.

As stated earlier, it is important to have typed offers. However, dealing with software quality, the CosTrading service lacks a standard language to describe offer types. The only available means is to use the `add_type()` operation of the `ServiceTypeRepository` interface provided by the CosTrading service. Section 3.2 discusses how the

```
module CosTrading {
  interface Lookup : TraderComponents,
                     SupportAttributes,
                     ImportAttributes {
    void query (
      in ServiceTypeName  type,
      in Constraint       constr,
      in Preference       pref,
      in PolicySeq        policies,
      in SpecifiedProps   desired_props,
      in unsigned long    how_many,
      out OfferSeq        offers,
      out OfferIterator   offer_itr,
      out PolicyNameSeq   limits_applied
    ) raises (
      IllegalServiceType,
      UnknownServiceType,
      IllegalConstraint,
      IllegalPreference,
      IllegalPolicyName,
      PolicyTypeMismatch,
      InvalidPolicyValue,
      IllegalPropertyName,
      DuplicatePropertyName,
      DuplicatePolicyName
    );
  };
};
```

Figure 3: The Lookup Interface to Search Offers.

TORBA Definition Language (TDL) addresses this problem.

2.3 Searching Service Offers

As this paper focuses on the search process, the drawbacks of the export process are not discussed here. However, these drawbacks are similar to those presented in this section.

Once offers have been exported by servers, their references and properties could be retrieved using the CosTrading search operation. Figure 3 presents its `Lookup` interface used to perform searches. The `query` operation allows clients to find back services from the set of exported offers. The argument number of this operation is quite high. This is due to the genericity required by the operation in order to be usable in a wide number of applications.

The `type` parameter defines the offer type required by the client application. The `constr` parameter contains a constraint to be matched by the properties of selected offers. This constraint is a string containing a boolean expression written using the *OMG Constraint Language* (OCL). The `pref` parameter specifies the returned offer order in OCL. The `policies` parameter specifies the strategies to be used during the search. The `desired_props` argument contains the properties to be returned for each offer to the client: none, all, or only specific ones. As there may be a huge number of matching offers, the `how_many` argument fixes the maximum number of offers to be returned. Following offers could be retrieved later on using the `offer_itr` iterator provided by the operation. Finally, the two last `out` parameters contain, after processing, the result (`offers`) and the limits effectively applied to the search policy (`limits_applied`).

Furthermore, when providing wrong parameter values, the `query` operation raises one of the ten listed exceptions. Such exceptions mean a misuse of the CosTrading service related to search strategies or to its type model, like an illegal or unknown type name, a badly expressed preference or constraint, or an unknown property name. Thus, the CosTrading service only checks requests at runtime, while type checking could be performed at application compilation time, improving both software quality and service performance—avoiding runtime type checking. At the moment, TORBA, as discussed in the following, mainly addresses type checking at application compilation time.

Figure 4 presents how an application, written in OMG IDLscript[1] [21], may retrieve offers about color printers faster than two pages per minute. The `offers`, `iter` et `limits` variables are initialized to receive the `query()` operation results. The offer type, property constraints, and the result order are provided as strings. Thus, it is up to the CosTrading server to check and to evaluate these strings in order to perform the search, implying type errors to be only discovered at runtime.

The simplicity of this excerpt relies on the use of the OMG IDLscript language. However, a real application has to set the search strategy, catch and process the potential exceptions, and process the returned results. The latter includes the `offers` sequence processing, and potentially the use of the `iter` iterator to process the following offers. Thus, about fifty lines of Java or C++ are required only to ob-

[1] IDLscript is the CORBA 3.0 scripting language, contribution and specification of our project CorbaScript [4, 13, 14].

```
# variables to receive answers
# using the out mode
offers = Holder()  # returned offers
iter   = Holder()  # next offers iterator
limits = Holder()  # limits applied
trader.query (
  "Printer",       # offer type required
  # OCL for offer constraint
  "color == TRUE and ppm > 2",
  "first",         # answer order
  # use default strategy
  CosTrading.PolicySeq(),
  # properties to be returned
  CosTrading.Lookup.SpecifiedProps (
    CosTrading.Lookup.HowManyProps.some,
    ["queue", "color",
     "cost_per_page", "ppm"]
  ),
  # max number and out params
  100, offers, iter, limits
)
```

Figure 4: Searching offers using IDLscript.

tain the list of color printers faster than two pages per minute. In that, we claim that the `query` operation is complex and very technical to use. Moreover, the huge use of this operation forces applications to build, invoke, and process many trading requests, introducing code complexity and potential runtime errors. Section 3.3 discusses how TORBA automates this trading related technical code production to simplify application code.

2.4 Review

More than presenting the main operations provided by the ODP/OMG CosTrading service to type and search offers, this section outline the drawbacks of these functions. First, the CosTrading service does not provide a definition language to define offer types. Such a language is mentioned in the CosTrading specification, however only for an illustrative purpose. The service only relies on the use of a type repository used at runtime. Then, the technicity and complexity of this service have been discussed. In order to benefit from the CosTrading service, it is necessary to master the use of operations like `query` and data structures provided by the service. Finally, using strings to manipulate properties implies runtime type checking and forbids type checking at compilation time. This reduces the easiness to produce reliable applications in an efficient way. To summarize, as any CORBA service, the CosTrading service only offers a set of complete OMG IDL interfaces. This brings the following four questions.

- How to simplify the use of the CosTrading service?
- How to provide type checking at compilation time?
- Which language should be used in order to define offer types?
- Which framework should be applied to trading?

Looking at today's software industry, three answers arise. First, a GUI could be provided to use the trading service easily. This solution is already available for many trading service products. Nevertheless, this choice does not address the use of the trading service from an application. Then, a library may hide the trading service complexity. However, providing such a library is a huge task: It would be easy to suffer the same drawbacks as the CosTrading interfaces. Moreover, it would only define a programming framework, but no design method, nor language to specify offer types. Finally, a trading function, to be specialized to each application needs, could be defined using the concept of *trading contract* as discussed in the following section.

3 The TORBA Proposal

3.1 The Trading Contract Approach

The objective of TORBA is to provide a simple and strongly typed trading facility for CORBA applications. In that, TORBA is based upon the ODP/OMG CosTrading service, taking full advantage of its functionalities like available implementations, complex lookup algorithms, offer persistence, large-scaled trader federations.

Then, the conceptual benefit of TORBA is to define the concept of *trading contracts*. Such a contract is defined at application design time like OMG IDL interface contracts are defined [9]. These contracts take into account offer provider needs as well as client application ones: This results in the definition of trading offer types. First, offer types clearly identify and group together properties (i.e. name and type of the values) characterizing exported CORBA objects conform to a given OMG IDL interface. Second, offer types also contain a list of query operations commonly used in client applications. Such operation is characterized through a synopsis (name and parameters), as well as a boolean constraint to be applied on both parameters and the properties of the associated type. Offer types may be classified using multiple inheritance. Such a classification permits designers to define abstract types, like a device, that could be specialized to concrete types, like a scanner and a printer. Moreover, concrete types can also be inherited to define new query operations exactly meeting requirements of client applications. Using multiple inheritance improves the reuseness of properties and query operations.

The technical benefit of TORBA is to provide a complete generation and execution environment to use trading contracts. Offer types are defined using the *TORBA Definition Language* (TDL). Such definitions are then compiled to generate trading proxies offering to applications easy-to-use OMG IDL interfaces. The use of these specialized interfaces is thus checked at application compilation time. Moreover, proxy implementations fully hide the ODP/OMG CosTrading technicity. Such implementation is generated for several programming languages: OMG IDLscript, Java, and C++ later on. In the meantime, trading contracts could also be used dynamically through a generic graphical console. Trading contracts are stored into a repository and browsed by the console which can also invoke query operations defined for the given type.

3.2 The TORBA Definition Language

The *TORBA Definition Language* (TDL) is the formalism to define TORBA trading contracts. Using simple typed constructions, it describes offer types, their inheritance relation, their properties (name and type), as well as query operations (name, parameters, and constraints). Property and parameter types rely upon the OMG IDL type model. Constraints are defined using the OMG Constraint Language (OCL) extended to take into account query operation parameters as well as to offer composition of query operations. TDL is defined as two languages: an XML DTD and a BNF grammar. This paper only describes the second one, being more concise and quite familiar to CORBA users. Figure 5 presents an example of offer type definitions.

```
abstract offer Device {
  property string name ;
  query all () is TRUE ;
};
offer Printer : Device {
  interface PrintService ;
  property boolean        color ;
  property float          cost_per_page ;
  property unsigned short ppm ;

  query colors () is color == TRUE ;
  query faster (in unsigned short s)
    is ppm > s ;
  query faster_colors (in unsigned short s)
    is colors () and faster (s) ;
};
```

Figure 5: Trading Offer Type Definition using TDL.

A trading offer type is defined using the `offer` keyword followed by the type name, and possibly the list of inherited super-types. Basically, a type is concrete: provider could export offers using this type. Then, it has to include an `interface` entry defining the base interface to be supported by exported objects. The `abstract` keyword defines a type as being abstract, no offer may be exported for this type. It will be inherited to define concrete types. The TDL contract of Figure 5 defines two offer types related to the printer example of this paper: The `Printer` concrete type inherits from the `Device` abstract type, and specifies offers for objects implementing the `PrintService` interface (or one of its sub-interfaces). Properties are defined using the `property` keyword followed by an optional access mode, an OMG IDL type, and a formal name. If undefined the access mode is `normal` (see section 2.2).

The `Printer` offer includes the four following properties: the `name` string inherited from `Device`, the `color` boolean, the `cost_per_page` float, and the `ppm` unsigned short. Search operations are defined using the `query` keyword followed by a name, potentially a list of arguments (defined as for OMG IDL operations), and a constraint. The constraint is based upon the properties of the offer type (e.g. the `colors()` query), the properties and the parameters (e.g. the `faster()` query), or a composition of query operations (e.g. the `faster_colors()` query). The `all()` query is defined with `TRUE` as constraint in order to retrieve all the available offers for the `Printer` type. Query operation inheritance has the following semantic: The constraint is kept, however it does not apply on the super-type, but on the inherited type. The operation implementation is implicitly overloaded in generated proxies. When applied to the `Device` type, the `all()` operation returns all the available device offers. When applied to the `Printer` type, it only returns the available printer offers.

Defining specific queries for given values of properties should not be misused. The point is not to define a query for any potential property value, but to define the most commonly used queries. For queries that may appear from time to time only, the generic query operation available with all types has to be used (see section 3.3.1). Nevertheless, using the generic `query()` generated by TORBA already brings type checking and reduces the technicity of the lookup mechanism.

Two constraints are implied by the use of TDL contracts. First, like OMG IDL contracts, TDL contracts have to be globally known to clients. Moreover, the type hierarchy of TDL may be extended but has to stay consistent, i.e. no TDL contract should be removed nor modified. Second, each TDL contract has to be defined using an identifier being unique in the whole system. Here too, like OMG IDL definition it is important for designers to define their TDL contracts using modules in order to avoid name collisions.

This section has presented the second TDL formalism (BNF grammar), being simple to learn. This basis will be extended according to the need arising from our experiments. As an example, dynamic properties specification, whose values are computed at runtime and not statically set at exportation time, seems an interesting extension. However, it is important for this language not to become too complex and underused due to this complexity.

3.3 Trading Proxy Generation

Once trading contracts have been defined using TDL, they may be compiled to generate trading proxies for applications, as depicted in Figure 6.

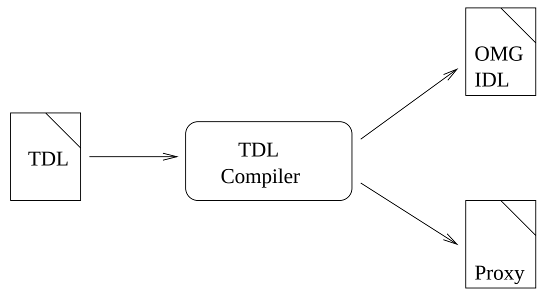

Figure 6: TDL Language Compilation Process.

The TDL compiler checks both the syntax and the semantic of TDL definitions. Semantic checking controls OMG IDL type correctness, TDL type names and properties, as well as OCL constraints in order to ensure no type related problem could arise at runtime. Proxy OMG IDL interfaces are produced by the TDL compiler, as well as their implementation for a given language—IDLscript and Java for the moment, C++ later on. For portability purpose, the TDL compiler is written in the Java language, based on the lexical and syntactic analyzer generated using JavaCC [15].

3.3.1 Generated OMG IDL Interfaces

Each definition of trading offer type is mapped to an OMG IDL module named as the offer type and containing the five following definitions.

- The `Offer` structure represents a trading offer. It contains the exported object reference and a field for each property defined in the offer type or its super-types. Field types are those of the service interface and property types as defined in the TDL offer.
- The `OfferSeq` sequence is used by query operations to return matching offers.
- `OfferType`, `Export`, and `Lookup` interfaces respectively describe the **Service Type Repository** access, the export and the lookup proxies. The latter inherits from the `TORBA::Lookup` interface and contains an operation for each query definition. Its also contains a generic but nonetheless typed query operation.

Figure 7 presents an excerpt (the lookup proxy interface) of the OMG IDL definitions generated for the `Printer` trading contract as defined in Figure 5.

```
#include <TORBA.idl>
module Printer {
  struct Offer {
    PrintService   service ;
    string         name ;
    boolean        color ;
    float          cost_per_page ;
    unsigned short ppm ;
  };
  typedef sequence<Offer> OfferSeq ;

  interface Lookup : TORBA::Lookup {
    OfferSeq query_all () ;
    OfferSeq query_colors () ;
    OfferSeq query_faster
      (in unsigned short s) ;
    OfferSeq query_faster_colors
      (in unsigned short s) ;
    OfferSeq query (in TORBA::Query q)
      raises (TORBA::IllegalConstraint) ;
  };
  // interfaces for type definition
  // and exportation
};
```

Figure 7: OMG IDL Module Generated from the Printer TDL Contract (excerpt)

The `Printer` offer type is mapped to the `Printer` OMG IDL module. The `Offer` structure represents a printer offer. It contains a field for the exported print service, as well as for the `name`, `color`, `cost_per_page`, and `ppm` properties. The lookup proxy `query_all()`, `query_colors()`, `query_faster()`, and `query_faster_colors()` operations represent the queries defined in the `Printer` contract. Parameters are the same as those defined in the contract, while their return type is a printer offer sequence (i.e. `OfferSeq`). The last `query()` operation allows applications to perform searches not defined in the TDL contract. The `TORBA::IllegalConstraint` exception may be raised at runtime if the constraint is malformed.

Experiments have been performed using generation rules presented here, validating these choices. As an example, the `Offer` structure is a good means to perform checking of export and lookup operations at compilation time. However, we also intend to experiment the use of *valuetypes*[2] instead of the structure, as well as using a typed iterator interface instead of the sequence.

[2]Since CORBA 2.3, *valuetypes* permit argument objects to be passed by value instead of by reference.

```
# File 'PrinterProxies.cs'
import TORBArt
class Lookup (TORBArt.LookupBase) {
  proc __Lookup__ (self) {
    self.__LookupBase__ ("Printer",
                         Printer.Lookup)
  }
  proc query (self, constraint) {
    answers = []
    for offer in
      self.generic_query (constraint) {
      answers.append (
        Printer.Offer (
          offer["service"],
          offer["name"], offer["color"],
          offer["cost_per_page"],
          offer["ppm"]
      ) )
    }
    return answers
  }
  proc query_faster (self, s) {
    return self.query ("ppm >= " +
                       s._toString())
  }
  proc query_all (self) {
    return self.query ("TRUE")
  }
  # other query operations
}
```

Figure 8: OMG IDLscript implementation of the Printer lookup proxy (excerpt)

3.3.2 Generated Proxy Implementation

The generation of proxy implementations depends on the constructions of a given language. However, using an object-oriented language, each OMG IDL interface is implemented by a class inheriting from a base class provided by the TORBA runtime. These classes fully hide the ODP/OMG CosTrading technicity: use of the service interfaces and data structures, as well as exception handling. Such runtime classes provide generic operations used from proxy implementations. Figure 8 presents an excerpt of the lookup proxy implementation for the `Printer` offer type, generated for the OMG IDLscript language.

The `Lookup` class inherits from the `TORBArt.LookupBase` class provided by the TORBA runtime. The `__Lookup__` constructor invokes the super-class constructor providing the TDL type name (i.e. `Printer`), as well as the im-

```
lookup = PrinterProxies.Lookup()
offers1 = lookup.query_faster_colors (2)
offers2 = lookup.query ("color == FALSE
  and cost_per_page < 0.05 and ppm > 10")
```

Figure 9: Printer Search Proxy Use.

plemented interface type (i.e. `Printer::Lookup`). The generic query operation is invoked by the `query` operation providing the constraint to apply. Then, the result is translated to a printer offer sequence (i.e. `Printer::OfferSeq`). The implementation of query operations, like `query_faster` and `query_all`, only consists of creating the associated constraint and invoking the `query` operation.

3.4 Using TORBA Proxies

Figure 9 presents, in OMG IDLscript, the use of lookup proxy presented in the previous section. The first line instantiates the lookup proxy class. The second line invokes the query operation to find color printers faster than two pages per minute. This operation realizes the same search processing as the one presented in Figure 4. Simplicity brought up by TORBA becomes clear. The application developer does not bother with the trader technicity, he/she can focus on the use of the trading contract only. Moreover, the operation execution cannot fail as types have been checked by the TDL compiler. It can only return an empty sequence if no offer matches the search. The third line illustrates the option of using a search operation not defined in the trading contract: Searching offers related to B&W printers faster than ten pages per minute, for a cost less than five cents a page. Nevertheless, even if the use of this operation is provided, software engineering quality is improved when all the search requests are defined in the trading contract.

3.5 Execution of TORBA Proxies

Figure 10 presents the set of objects involved during the execution of a query operation. The lookup proxy object and its CORBA stub are colocated with the application. This latter invokes the proxy operations through its OMG IDL interface. The proxy operation implementation invokes the TORBA runtime class providing the appropriate constraint. Then, this class invokes the CORBA stub providing access to the ODP/OMG CosTrading service. As a result, the runtime class catches the exceptions and the proxy class translates data from its CosTrading representation to the representation defined in the trading contract. One future work is to measure the overhead of lookup proxies and to optimize their implementations in order to be close to native CosTrading performance (see section 4).

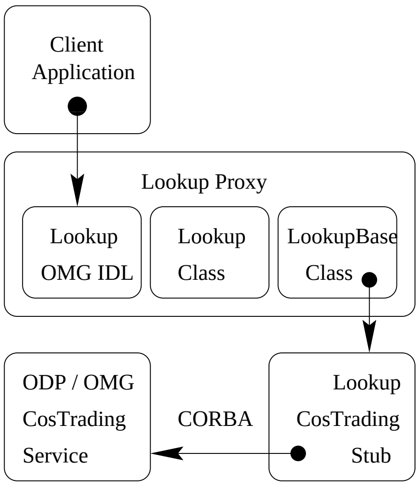

Figure 10: Execution Process of Lookup Proxy Operations.

3.6 The TORBA Dynamic Approach

Previous sections have presented the conceptual benefits of TDL contracts, as well as the technical ones brought by generation and execution of related proxies. In the meantime, the TORBA environment offers a dynamic approach to use trading contracts as depicted in Figure 11. This approach permits one to build applications without static knowledge, at design time, about used trading contracts. This knowledge will be learnt at runtime.

The dynamic approach in TORBA relies on a trading contract repository. This repository, currently written in OMG IDLscript, stores TDL contracts as a graph of CORBA objects. Each object of the graph represents at runtime a semantic construction of the TDL language. Thus, `offer`, `property`, and `query` constructions are mapped to `OfferDef`, `PropertyDef`, and `QueryDef` interfaces defined in the `TORBA` module. These interfaces provide operations to create and browse objects of the graph,

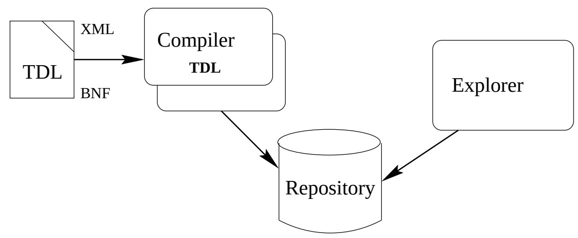

Figure 11: The TORBA Dynamic Approach.

providing TDL information at runtime. Creation operations are used by a specific version of the TDL compiler in order to feed the repository. Other operations are used by any TORBA application requiring dynamic discovering of available trading contracts. In order to validate this approach, we have realized in JavaIDLscript[3] a first dynamic application: the TORBA explorer, illustrated in Figure 12.

Through a GUI written using Java Swing, the TORBA explorer allows users to browse available trading contracts, to select a contract, to consult associated offers, and to perform predefined or specific query operations. The explorer implementation does not rely on any trading contract: Graphical interfaces are dynamically built at runtime according to trading contracts discovered into the TORBA repository. Thus, the TORBA explorer provides a trading GUI dedicated to the contracts used by applications, unlike GUI included with CosTrading implementations. Finally, this explorer is a generic and graphical proof of the relevance and strength of the trading contract concept presented in this paper.

4 Empirical Results

TORBA introduces extra processing while sending requests to the CosTrading. This implies an overhead. However, in the context of distributed applications, there are two levels in the evaluation of overhead. First, there are remote method invocations which overhead is potentially high. Second, there are local method invocations which overhead is most of the time insignificant compared to remote method one. In the context of TORBA, the overhead is introduced by the use of a local library, which means local invocations only : Three local invocations are added for each trading request. Thus, it just increases local processing time and keeps the number of remote method invocations identical, compared to the standard use of the CosTrading. Then, based on early test performed using the ORBacus Trader, the overhead introduced by TORBA is, without any ORB specific optimizations in producing TORBA proxies, less than 5%.

There are few ways to improve performance related to trading using TORBA. First, if the only use of the trader is performed through TORBA, then most of the trader checks (like type checking) can be removed since already performed by the proxies. Thus, the overhead brought up by the proxy would be balanced. Second, proxies could be located close to the trading server and not close to the client. Then, the number of network requests could be optimized, improving global performance. Third, using *smart proxies*, local to the client, to perform caching of trading results, the global performance could be optimized. Finally, a composition of the three aspects will bring the best results. Points two and three are not incompatible as proxies would be split between the trading server and the client application. Client side proxies would perform caching while server side ones would be dedicated to type checking and network optimization.

5 Comparison and Source of Inspiration

During the 80's, the ODP community has defined a type management system for the ODP trading function [10]. This research result has been partly integrated in the ODP / OMG CosTrading specification. However, to our knowledge, no similar works to our TORBA proposal have been performed to reduce the CosTrading complexity and to increase reliability of trading based applications. The trading aspects of ODL (Object Definition Language) defined by the TINA consortium [22] were only related to defining trading properties of objects. This could only be seen as basic trading contracts : attributes are inevitably strings and ODL do not permit to define typed queries. Thus, the complexity of the trader's use is not reduced actually. In the meantime, our original work relies on the use of well-known mechanisms of distributed object computing

[3] JavaIDLscript is our second implementation of the OMG IDLscript language offering access to CORBA and Java objects in the meantime.

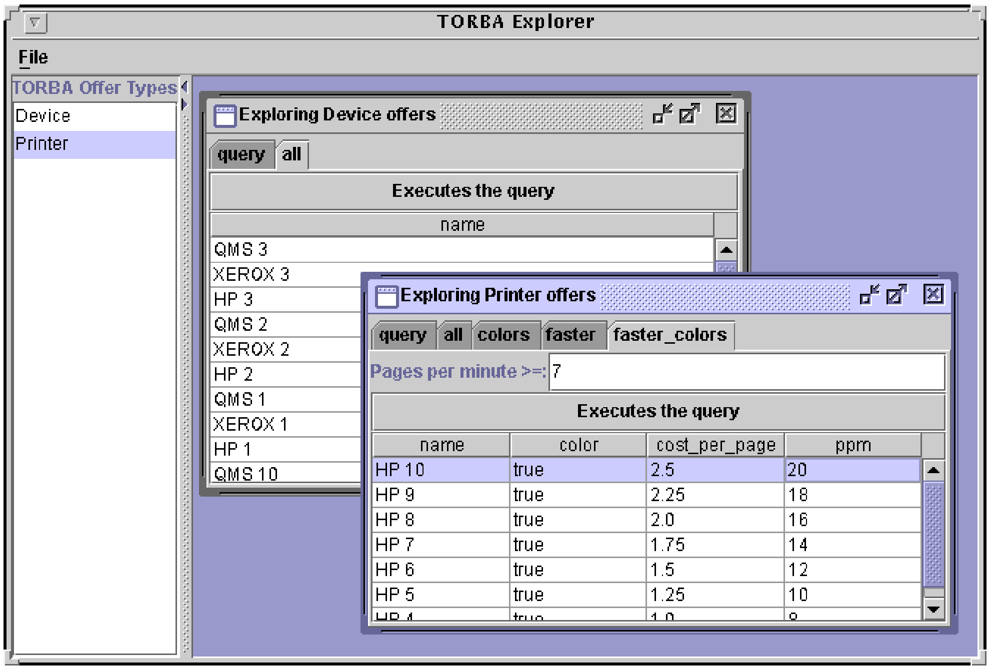

Figure 12: The TORBA Explorer.

middleware: the proxy principle, the ORB structure, and the component approach.

The proxy principle has been defined in [23] as a structural concept to build distributed applications, acting on the behalf of a remote object. This principle extends the RPC (Remote Procedure Call) mechanism as defined in [3] in order to use it in an object-oriented context (i.e. Remote Method Invocation). At the communication level, a proxy (a.k.a. stub) serializes invocations to remote objects like in CORBA [19], DCOM [8], and Java RMI [24] environments. Such a proxy implementation fully hides the technicity related to the serialization process: marshalling of parameters into a network message, care taking of heterogeneity, network layer and error management, and finally unmarshalling the network reply to application data. These proxies are generated based on communication contracts written using an interface definition language (IDL). These IDL descriptions simplify and bring automation to produce the implementation of communication means, increasing the reliability of applications. In the context of TORBA, the communication contract concept, the IDL language, and communication proxies are transposed to trading contracts, the TDL language, and trading proxies. Thus, TDL descriptions simplify and bring automation to produce code related to trading, increasing application reliability.

Compared to *smart proxies* used in the Quality of Objects (QuO) middleware [30], or as implementation of meta-programming mechanism [29], TORBA proxies cannot be labeled as *smart*. In these two examples, *smart proxies* are proxies that potentially perform more processing—like logging, caching, QoS control, or meta-programming—in a transparent way from the client point of view. Extra processing is added to the standard one, without modifying the proxy interface. In the context of TORBA, proxies have an explicit interface which is different from the classical interface of the CosTrading. Moreover, *smart proxies* tend to offer dynamic mechanism for reconfiguration while TORBA proxies are quite static, and could not be changed dynamically at runtime.

TORBA is close to CORBA. The OMG IDL language permits designers to describe interface contracts for CORBA objects, while the TDL language permits them to define trading contracts. The OMG IDL language is compiled to produce communication stubs, or to feed the Interface Repository. Similarly, the TDL language is compiled to generate trading proxies, or to feed the trading contract repository. CORBA stubs rely on an ORB runtime, encapsulating the GIOP/IIOP protocol, while TORBA proxies rely on the TORBA runtime hiding the CosTrading service, as well as the ORB.

The component-oriented approach is the last source of inspiration of the TORBA proposal. As an example, the CORBA Component Model [18] defines a component as being a software entity providing multiple interfaces (or facets). Each facet is a point of view on the component, which logically defines a set of operations. In that, TORBA provides access to the generic ODP/OMG trading service through facets dedicated to application requirements. Each lookup proxy generated is a dedicated facet being a point of view on the trading service as depicted in Figure 13.

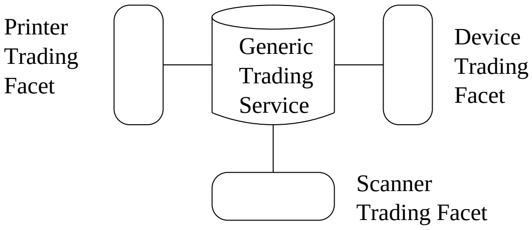

Figure 13: TORBA, towards a 'componentized' trading service.

6 Conclusion

First, this paper has reviewed the ODP/OMG `CosTrading` service. This review has presented the use of the service as being very technical and complex due to the lack of a structured approach. The various drawbacks brought up by the lack of type-checking at compilation time have been underlined. Then, the lack of formalism to define offer types and search operations has been presented as being one of the reasons of the service complexity.

Then, TORBA has been presented as a framework structuring the ODP/OMG trading service use. The conceptual contribution of this paper relies on the definition of the trading contract concept as a paradigm to structure the trading activity. The benefits of the TDL formalism use and its associated tools have been discussed. Using an example, the benefits of TORBA have been illustrated in terms of type checking, simplicity, productivity, and reliability of applications.

All the elements depicted in this paper have been prototyped and experiments have been performed using IDLscript and Java languages, as well as the ORBacus trading service [20]: TDL compilers (BNF and XML versions), proxy generators (OMG IDL, OMG IDLscript, and Java), runtime environments for IDLscript and Java, the trading contract repository, as well as the TORBA explorer are already operational. The next step is to finalize the TORBA environment in order to release it, and to obtain experiment/use feedback from end-users.

From now on, we have lots of work in view around TORBA: (1) support of C++ applications, (2) experiments over other `CosTrading` implementations, (3) measure of the overhead implied by TORBA proxies, (4) experiments of iterators, dynamic properties, and lookup strategies, (5) extension towards asynchronous trading (notification to applications of newly exported offers), and (6) use of the TORBA approach in the context of Jini, trading serialized objects and not only references.

In the meantime, TORBA is part of our actual research work. We intend to use TORBA in order to experiment the concept of **Component Oriented Trading** (COT) [27]. In that, TORBA would become the basis of TOSCA (*Trading Oriented System for Component-based Applications*), whose goal is to provide an environment to deploy and to administrate distributed component based applications [12].

Finally, in a more ambitious vision, we intend to consider the benefits of a language to perform queries and to act upon distributed objects. The goal would be to unify search operations on trading services, object-oriented databases, and object environments *à la* JavaSpaces [25]. This language could be named **TORBA Query Language**, relying upon the following equation:

$$TQL = TDL + OCL + OQL + IDLscript$$

References

[1] K. Arnorld and al. *The Jini Specification*. Addison-Westley, first edition, June 1999. *ISBN: 0-201-61634-3*.

[2] D. Belaid, N. Provenzano, and C. Taconet. Dynamic Management of CORBA Trader Feder-

ation. In *Proceedings of the 4th USENIX Conference on Object Oriented Technologies and Systems (COOTS'98)*, Santa Fee, New Mexico, USA, April 1998. USENIX.

[3] A. Birrell and B. Nelson. Implementing Remote Procedure Call. Technical Report CSL-83-7, Xerox, October 1983.

[4] CorbaWeb. CorbaScript Home Page. URL: http://corbaweb.lifl.fr.

[5] G. Craske and Z. Tari. A Property-based Clustering Approach for the CORBA Trading Service. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Las Vegas, 1998.

[6] G. Craske, Z. Tari, and K. Kumar. DOK-Trader: A CORBA Persistent Trader with Query Routing Facilities. In *International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, September 1999.

[7] J.-P. Deschrevel. The ANSA Model for Trading and Federation. Technical report, ANSA, July 1993.

[8] R. Grimes. *Professional DCOM Programming.* Wrox Press ltd., Birmingham, Canada, 1997.

[9] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++.* Addison-Westley, 1999. *ISBN: 0-201-37927-9.*

[10] J. Indulska, M. Bearman, and K. Raymond. A Type Management System for an ODP Trader. In *Proc. of the International Conference on Open Distributed Processing (ICODP'93)*, pages 141–152, Berlin, Germany, September 1993.

[11] IS0. *Open Distributed Processing Reference Model – parts 1-4.* International Standard Organization, 1995. ISO 10746-1..4.

[12] R. Marvie, P. Merle, and J.-M. Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, Antwerp, Belgium, September 2000. IEEE.

[13] P. Merle, C. Gransart, and J.-M Geib. CorbaScript and CorbaWeb: A Generic Object Oriented Dynamic Environment upon CORBA. In *Proceedings of TOOLS Europe 96*, Paris, June 1996.

[14] P. Merle, C. Gransart, and J.-M. Geib. Using and Implementing CORBA Objects with CorbaScript. *Object-Oriented Parallel and Distributed Programming*, 2000. Ed. Hermes.

[15] Metamata. Java Compiler Compiler User Guide. *http://www.metamata.com/javacc/index.html.*

[16] Y. Ni and A. Goscinski. Trader Cooperation to Enable Object Sharing among Users of Homogeneous Distributed Systems. Technical report, RHODOS Project, 1993.

[17] OMG. *CORBAServices: Common Object Services Specification.* Object Management Group, November 1997.

[18] OMG. *CORBA Components: Joint Revised Submission.* Object Management Group, August 1999. OMG TC Document orbos/99-07-{01..03,05} orbos/99-08{05..07,12,13}.

[19] OMG. *CORBA/IIOP 2.3.1 Specification.* Object Management Group, October 1999.

[20] OOC. ORBacus Trader. *http://www.ooc.com.*

[21] OOC and LIFL. *CORBA Scripting - Joint Revised Submission.* Object Management Group, August 1999.

[22] A. Parhar. TINA Object Definition Language Manual v2.3. Technical report, TINA-C, 1996.

[23] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS 86)*, pages 198–204, Cambridge, Mass., USA, May 1986. IEEE.

[24] Sun. *Java Remote Method Invocation Specification.* Sun Microsystems, October 1998.

[25] Sun. *JavaSpaces Service Specification.* Sun Microsystems, May 2000.

[26] Z. Tari and G. Craske. A Query Propagation Approach to Improve CORBA Trading Service Scalability. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, Taiwan, April 2000. IEEE.

[27] S. Terzis and P. Nixon. Component Trading: The Basis for a Component-Oriented Development Framework. In *WCOP'99 Proceedings of the Fourth International Workshop on Component-Oriented Programming*, 1999.

[28] A. Vogel, M. Bearman, and A. Beitz. Enabling Interworking of Traders. In *Proceedings of the 3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, February 1995. ChapMan and Hall.

[29] N. Wang, K. Parameswaran, and D. Schmidt. The Design and Performance of Meta-Programming Mechanism for Object Request Broker Middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio TX, USA, January 2001. USENIX.

[30] J. Zinky, D. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1), April 1997.