University of Alberta

Building an Interoperable Distributed Image Database Management System

by

Bin Yao

Technical Report TR 00-07
May 2000

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Abstract

The DISIMA project addresses the development of a distributed, interoperable image database management system enabling content-based querying. The aim of this project is to add an interoperability feature to the existing prototype.

As object-oriented distributed computing platforms, OMA and, in particular, CORBA, can be helpful for database interoperability in terms of managing heterogeneity of platform and communication levels. The complete distributed architecture will involve homogeneous systems and heterogeneous systems. In this project, the goal is to develop a middleware on top of CORBA for homogeneous systems. In the simplest scenario, we can assume all the systems involved are DISIMA sites, and that they have the same schema with different database instances.

This technical report thus describes the development of such simpler (but not trivial) scenario. We also highlight and justify some design decisions made as the project was developed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Scope

1.1.1 The Content-based Image DBMS

An *information system* is a computer software developed to manage and manipulate data (mostly textual data), relevant to a certain application domain, for example: Management Information System (MIS) or Geographical Information System (GIS). In the last three decades, *database management systems (DBMSs)* have proven to be efficient tools to deal with such data. Some well-known commercial DBMS products are *Oracle* [19], *IBM DB2* [14], *Sybase* [42], *Microsoft SQL Server* [17], and *Informix* [16]. While traditional DBMSs handle only textual data, the recently expanding use of multimedia data (e.g., still images, video, and audio) in information systems has raised the demand for efficient management of other types of data, especially images.

An image DBMS is a software system that enables the acquisition, storage, manipulation, and retrieval of large amounts of image data and related symbolic data. Due to the visual nature of images, the use of traditional keywords or text annotation matching to query images is inadequate. More efficient and effective image retrieval methods need to be based on image visual features, such as color, texture, shape, and spatial relationship, as well as content semantics. This is referred as *content-based image retrieval* [4].

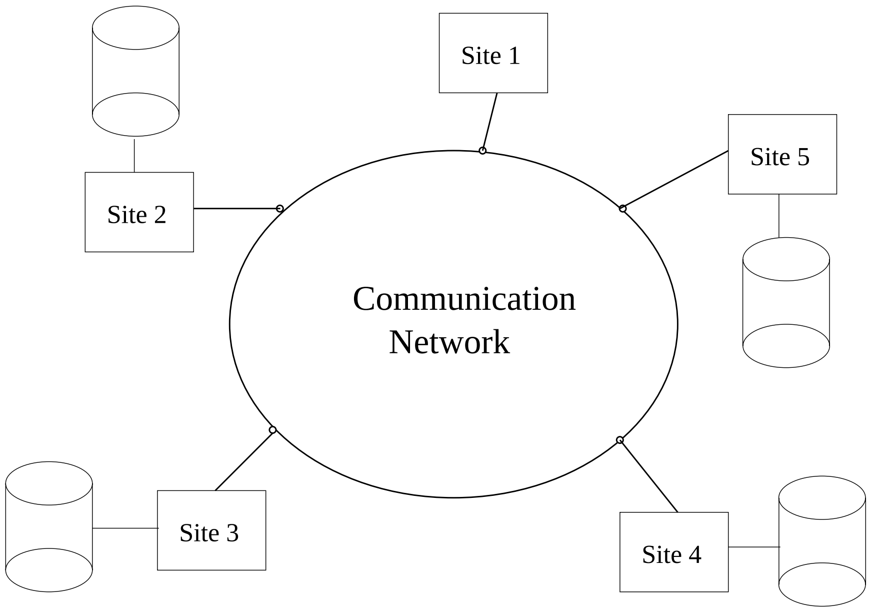


Figure 1.1: The Distributed Database System Environment (Taken from [38])

1.1.2 The Interoperable Distributed DBMS

The development of computer network technology, especially the rapid growth of the Internet, makes distributed computing possible. During the last couple of years, more and more information systems have been using distributed data processing technology, i.e., distributing data among some autonomous and heterogeneous repositories. Most likely, these repositories are database management systems (DBMS), and the information systems are referred as distributed database management systems (distributed DBMS). As defined in [38], a *distributed database* is a collection of multiple, logically interrelated databases distributed over a computer network. A *distributed DBMS* is a software system that permits the management of the distributed database system and makes the distribution transparent to the users. Figure 1.1 depicts an example of a distributed database system environment.

The promises of distributed DBMSs include transparent management of distributed and replicated data, reliability through distributed transactions, improved performance, and easier system expansion. Detailed discussion on distributed DBMSs can be found in [38]. Distributed DBMSs bring us many advantages in addition to the potential heterogeneity both in hardware and software levels. Therefore, one important issue of distributed DBMSs that needs to be solved is how to manage the heterogeneity to provide *interoperability* to users in such environment.

It is well known that computer networks are usually *heterogeneous*. For instance, the internal network of a department at a university might consist of multiple computing platforms. There might be a mainframe that handles large scale computing, UNIX workstations for daily research use, personal computers that run Windows/MacOS and provide desktop office automation tools, and other specialized systems, such as X terminals, telephony systems, and routers. A subsection of a given network may be homogeneous, but the larger a network is, the more diverse its components are likely to be.

A direct result of heterogeneous platforms is that the software on top of these platforms has to cope with heterogeneity, in addition to all the problems normally encountered in distributed systems (e.g., problems associated with network resource sharing or the failure of some of the systems in the network). It is possible to end up with more than one version of an application for each platform, which makes maintenance more difficult. [21]

Note that, besides computing hardware and operating systems, heterogeneity in this context refers to the software (information system) itself as well. Different DBMSs may have different data models, different query languages, or different transaction management protocols — any of which can lead to heterogeneity. Furthermore, in many cases, there is still a large amount of data that is stored in non-DBMS repositories, such as file systems.

A feasible approach to achieving interoperability for distributed DBMS is using a *multidatabase* approach and *object oriented* technology [1]. A multidatabase system resides on top of an existing database system and provides the users with a single database interface including one global integrated schema and a unique query language. Object orientation, a complement to the multidatabase approach, has two basic features that are especially important as far as interoperability is concerned. The first one is *encapsulation* capability, which makes it possible to encapsulate existing DBMSs with different interfaces and implementations (even though some of them may not be actually DBMSs), and provide a common DBMS-like interface to the rest of the system. What has been proposed is also called a *wrapper*. Another property of object orientation that is useful here is *specialization/generalization*, which enables the abstracting of the similarities in entities coming from different databases during database schema integration. (See [38] for examples.)

Taking all of the above characteristics of information systems into account, there are some distributed object computing platforms which can facilitate the development of open systems, especially interoperable distributed DBMSs. The two most popular platforms are the *Object Management Architecture (OMA)* [27] from *Object Management Group (OMG)* and the *Distributed Component Object Model (DCOM)/Object Linking and Embedding (OLE)* [18] environment from Microsoft. Both of these platforms provide an infrastructure that support distributed objects communicating with each other, and both provide standard services that are commonly needed by all distributed components. Of these two platforms, OMA is superior to DCOM/OLE for its better interoperability (with respect to cross-language support, cross-platform support, network communications and common services); reliability (transactions, messaging, and security); performance (scalability); and viability (product maturity). (See [12] for a detailed comparison.)

1.2 Objectives

The DISIMA (see Chapter 2) project aims at the development of a database management system for images, which provides users with uniform interfaces to access multiple, distributed, and possibly heterogeneous image and spatial repositories.

As described in [24, 46], the features of the DISIMA project can be summarized as follows:

1. To use an object-oriented approach to build a DBMS kernel that provides flexibility for user-defined classification of images, provides support for feature-based and spatial querying over image content, and enables reasoning over spatial relationships for query optimization;

2. To use image processing and indexing techniques for efficient querying and access to image databases, and to develop query languages and primitives for querying image databases;

3. To provide scalability, interoperability, and open access to image repositories.

The single site prototype which accomplished the first two features has been implemented on top of a commercial Object Oriented DBMS (OODBMS) – ObjectStore. Details on the implementation of the DISIMA kernel and the query system can be found in [7]. This work focuses on the third feature.

The interoperable architecture used in this project is designed on top of a distributed object-oriented computing platform, CORBA, as defined in the OMA (see Chapter 3 for a detailed description of CORBA). The CORBA provides transparencies at the platform and communication levels. There remain two other levels — the database level, where different data models can be found, and the semantic level, where homogenization of the meanings of the objects takes place. The complete distribution architecture will involve both homogeneous and heterogeneous systems.

The objective of this project is to develop a middleware on top of CORBA for homogeneous systems. In the simplest scenario, we can assume all the systems involved are DISIMA sites, and that they have the same schema with different database instances. A query in this environment has to be sent to all the systems involved, and the query result is the union of the results from each of the systems.

This technical report thus describes the development of such simpler (but not trivial) scenario. We also highlight and justify some design decisions made as the project was developed.

1.3 Report Organization

This technical report includes six chapters followed by the Bibliography and Appendix.

- Chapter 1 is the introduction, which describes the background knowledge and defines the objectives of our research.

- Chapter 2 gives a brief description of the DISIMA project, including the DISIMA model, the DISIMA architecture, and its query language, MOQL and VisualMOQL.

- In Chapter 3, we review the OMA, a distributed object oriented computing platform provided by the OMG. Specifically, we describe CORBA and CORBAservices as well as a fully CORBA compliant implementation, MICO.

- Chapter 4 explains the CGI (Common Gateway Interface) technology, its advantages and disadvantages.

- The design and implementation of the distributed DISIMA system is presented in Chapter 5. We start from a single site DISIMA, and then build a

CORBA-based single site DISIMA to the distributed DISIMA. The implementation issues are discussed at the end of the chapter.

- Finally, Chapter 6 concludes the project and suggests possible improvements and future work.

Chapter 2

DISIMA System

The *DIStributed Image database MAnagement (DISIMA) System* research project was carried out by the Database Systems Research Group at the University of Alberta. The *Natural Sciences and Engineering Research Council (NSERC)* of Canada provided the funding of the project through a strategic grant.

In an image DBMS, users want to query images using image content, as well as conventional textual information. Content-based indexing is required to facilitate content-based image querying. Since the DISIMA project addresses both image and spatial databases, the DBMS also has to deal with more structured spatial related information, such as geo-referenced entities, attributes or specific properties of entities, and relationships between entities. The query language must be sufficiently sophisticated to allow content-based images similarity search, and also support high level temporal/spatial notions and relationships. Such a DBMS can be used in many application domains, e.g., office automation, education, medical and healthcare, and telecommunication.

The following sections give an overview of the DISIMA model, the DISIMA architecture, and the *MOQL* query language, as well as the *VisualMOQL* interface. Details on the DISIMA model and architecture can be found in [47, 48]. MOQL and VisualMOQL are fully described in [23] and [45, 46, 49], respectively.

2.1 The DISIMA Model

The DISIMA model [48] provides efficient representation of images and associated meta-data to allow a wide range of content-based queries, by using the concept of

Salient Object. As illustrated in Figure 2.1, the DISIMA Model comprises two main blocks: the image block and the salient object block. A *block* is defined as a group of semantically related entities.

2.1.1 The Image Block

The image block is composed of two layers: the *image* layer and the *image representation* layer. An image is distinguished from its representations in order to maintain an independence between them. This is referred to as *representation independence*. In the image layer, users define an image type classification to categorize images according to functional relationships between images.

Figure 2.2(a) shows an example of an image class hierarchy. The *Image* class is categorized into two classes, *EducatonalImage* and *NewsImage*, according to certain criteria. The *NewsImage* class can be specified by three sub-classes: *PoliticalImage*, which includes all images related to politics; *ShowbizImage*, which identifies images associated with show business; and *MiscImage*, which refers to all other images.

There are two major image representation modes: the *raster* (good for image applications), and the *vector* (useful for spatial applications).

2.1.2 The Salient Object Block

The definition of salient objects for a certain application can result in a type hierarchy, as shown in Figure 2.2(b). DISIMA presents the content of an image by a set of salient objects (i.e., interesting entities in the image) with certain spatial relationships to each other. The *salient object* block is designed to handle salient object organization.

DISIMA distinguishes two types of salient objects: physical and logical. A *logical salient object (LSO)* is an abstraction of a salient object that is relevant to some application. It retains image-independent generic information about this object of interest. A *physical salient object (PSO)* is a particular instance of this object which may appear in specific images. There is a set of information related to the physical salient object, such as the location and the shape of the object in that particular image. Obviously, a logical salient object can exist independently of images while a physical salient object exists only if the image in which the physi-

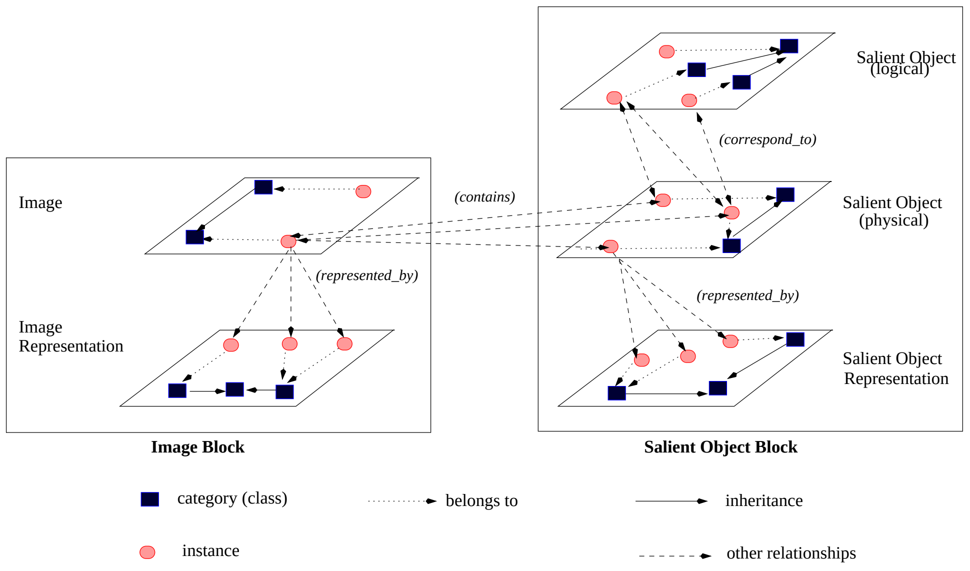


Figure 2.1: The DISIMA Model Overview (Taken from [47])

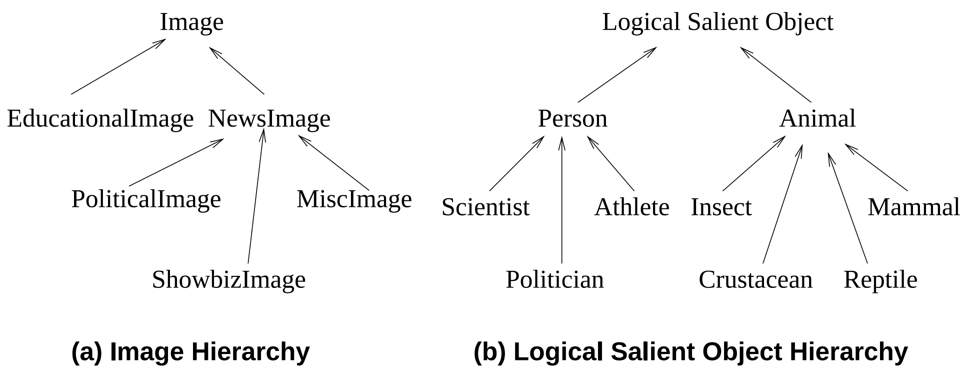


Figure 2.2: An Example of Logical Salient Object and Image Hierarchy

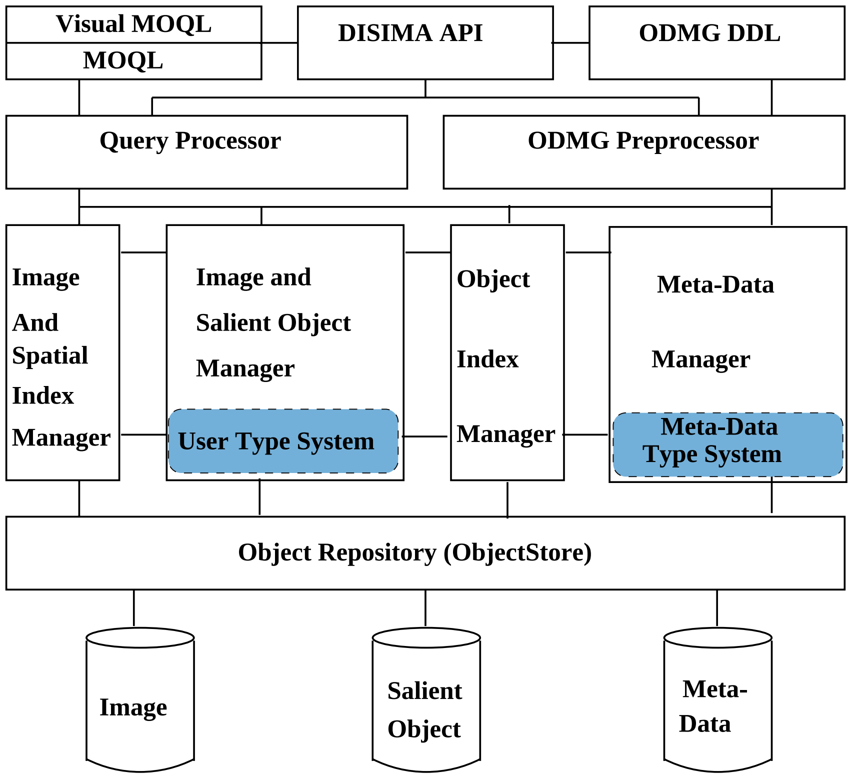


Figure 2.3: The DISIMA Architecture (Taken from [7])

cal salient object appears exists in the database. Also, there can be several physical salient objects linked to one logical salient object, since a salient object may appear in many images.

As in the case of the image block, the content information of salient objects is also separated from the representation. The representation can be either raster, which is used to access part of images, or vector, which fits well with spatial indexing and spatial relationships computation.

2.2 The DISIMA Architecture

As illustrated in Figure 2.3, The DISIMA architecture [48] comprises the interfaces, the processors, the meta-data manager, the image and salient object manager, the image and spatial index manager, and the object index manager.

The interfaces provide several ways (visual and alphanumeric) to define and query image data. The *VisualMOQL* [49] provides a user friendly graphical query interface, generating a query in the underlying query language *MOQL* [23], which is an extension of OQL [6]. (Query language aspects are discussed further in Section 2.3.) The DISIMA project uses the *data definition language (DDL)* from the *Object Data Management Group (ODMG)*. *DISIMA API* is a library of low level functions that allows applications to access the system services.

DISIMA is built on top of ObjectStore, which is used as an object repository. The *Image and Spatial Index Manager* and *Object Index Manager* are also included in the architecture because these object repositories may not have image and spatial indices. Even if they do, their indices may not match the DISIMA requirements. Furthermore, the image and spatial index manager and the object index manager allow dynamic index management (dynamic integration of new indices).

Although ObjectStore provides some querying facilities over collections, it does not have a built-in declarative query language and related query processor. Therefore, DISIMA has fully implemented MOQL queries and a *Query Processor* that handles the parsing and execution of the queries, including a MOQL parser and a query engine. The MOQL parser checks the semantics and syntax of the external query, which is then converted into an internal query tree with all the information given by the external query. Based on the internal query, the query engine generates an execution plan. Details can be found in [7].

The *Meta-Data Manager* handles meta information about images and salient objects. Based on object-oriented concepts, the DISIMA model integrates the raw image and its meta-data (the alphanumeric data linked to it). Meta-data is a kind of on-line documentation. It is important in improving the availability and quality of the information to be delivered.

The *Image and Salient Object Manager* is the kernel part of the DISIMA model. The *User Type System* (see [7] for details) defines the data structures and methods that support database population and image retrieval by similarity match. Images and salient objects are derived from a set of root types that are defined in the user type system. The image root type facilitates the recognition of salient objects appearing in images. The recognition is done semi-automatically. The image and salient object manager can control an application running in a classical transaction mode with the ACID properties, i.e., Atomicity, Consistency, Isolation, and

Durability.

DISIMA also supports an interoperable distributed architecture to allow users to query multiple image sources through a uniform interface. The architecture is designed on top of the Object Management Architecture (OMA) (see Chapter 3). The design and implementation of the architecture is the work of this project, and will be discussed in detail in Chapter 5.

2.3 Querying Images

2.3.1 MOQL

Object Query Language (OQL) from the Object Data Management Group (ODMG) is the general language for multimedia object oriented databases. It is an embedded language, which allows applications to query objects supported by the native language. OQL defines an orthogonal expression language in which all operators can be composed with each other, as long as the types of the operands are correct. The return object type of an OQL query may be inferred from the operators contributing to the query expression. The basic statement for querying objects in OQL is:

> **select [distinct]** projection_attributes
> **from** query [[**as**] identifier] {, query [[**as**] identifier] }
> [**where** query] [**group by** partition_attributes] [**having** query]
> [**order by** sort_criterion {, sort_criterion}]

Based on OQL with multimedia extensions, *Multimedia Object Query Language (MOQL)* is the query language used by DISIMA. The extensions include constructs to handle spatial, temporal, and presentation properties. Most extensions added to OQL by MOQL are in the **where** clause, in the form of four new predicate expressions: *spatial expression, temporal expression, contains predicate,* and *similarity expression* (see [23] for details). For example, the query "Find images with 2 people next to each other without any building, or images with buildings without people" can be expressed in MOQL as follow:

```
SELECT m FROM image m, building b1,
              person p1, person p2
WHERE ( m contains b1 and m not in
                (SELECT m1 FROM image m1, person p3
                WHERE m1 contains p3))
OR (m contains p1 and m contains p2
        and p1.MBB west p2.MBB and m not in
                (SELECT m2 FROM image m2, building b2
                WHERE m2 contains b2))
```

The above example shows that a MOQL query can be very complicated. Composing a MOQL query is not straightforward, even for a simple query. There is a need for a user friendly query interface that helps users easily construct MOQL queries.

2.3.2 VisualMOQL

A visual language uses graphical information to represent objects and the relationships among them. *VisualMOQL* is a visual query language that provides an easier way to compose queries, and then translates them into MOQL. It implements the image part of MOQL and allows users to query images by their semantics. Details can be found in [45, 46, 49].

Utilizing VisualMOQL, the user can query the database by specifying the salient objects in the image. The query can be refined by defining the color, shape, and other attribute values of these salient objects. The user can also specify the spatial relationships (both topological and directional) among salient objects in the image. Furthermore, the user can specify properties of the image metadata.

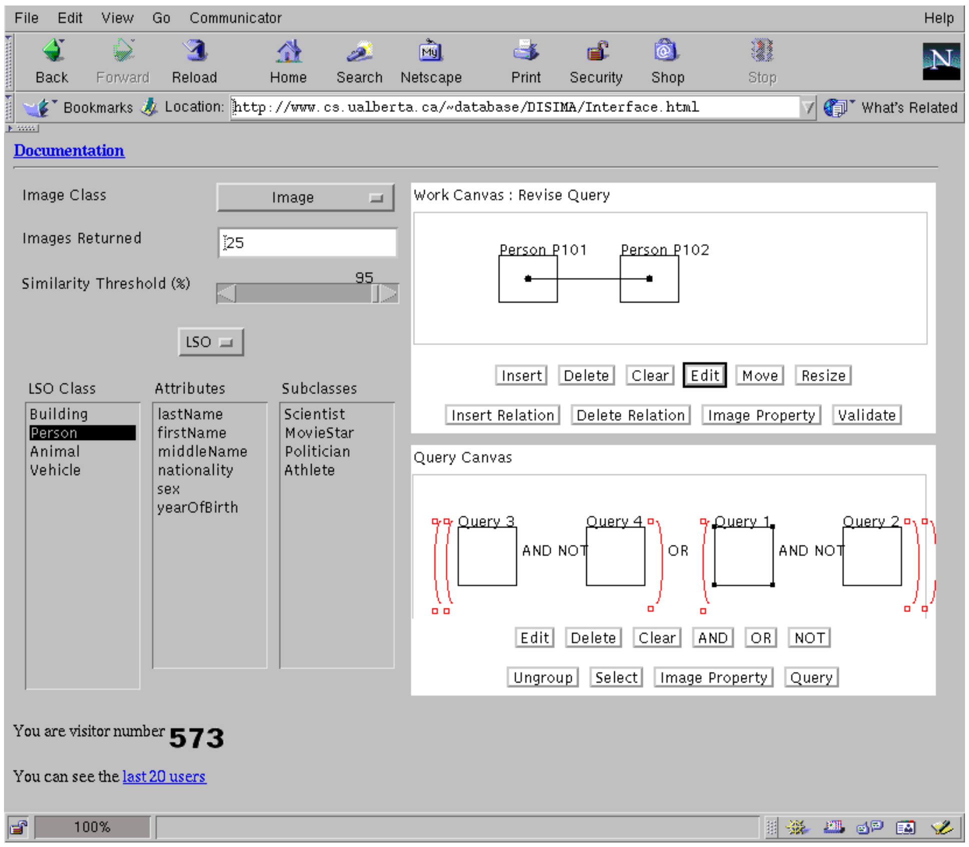
Described in [46], VisualMOQL has the following features:

- It is a declarative visual query language with a step by step construction of queries, which resembles natural language.
- It has a clearly defined semantics based on object calculus.
- It combines several querying approaches: semantic-based (query image semantics using salient objects), attribute-based (specify and compare attribute values), and cognitive-based (query by example).

VisualMOQL has been implemented as a platform independent *Java* applet. The user can access the VisualMOQL interface using any web browser (e.g., *Netscape Navigator*) on any platform, through the Internet. Figure 2.4 shows a VisualMOQL window which includes a number of components facilitating the composition of a query. The user can easily build a query by choosing the desired image class and the desired salient objects in the images. According to the type of query and the level of precision users want the result of the query to have, several levels of refinement are also provided. The VisualMOQL window is composed of:

- A chooser to select the image classes.
- A salient object class browser to choose the desired objects.
- An input field to specify the maximum number of images returned as the result of the query.
- A horizontal slider to specify the minimum similarity threshold between the query image and the target images stored in the database.
- A working canvas where the user constructs queries step by step.
- A query canvas where the user can compose compound queries based on subqueries defined in the working canvas, using AND, OR, and NOT operators.

The spatial relationship of two-person objects is illustrated on the working canvas of Figure 2.4. The expression of the query Q is shown on the query canvas. The VisualMOQL expression is then translated into MOQL (see Figure 2.5) before being submitted to the query processor. Figure 2.6 is a result window which includes a set of thumbnails of images matching the query. Clicking on the thumbnail returns the enlarged image.



Location: http://www.cs.ualberta.ca/~database/DISIMA/Interface.html

Image Class: Image

Images Returned: 25

Similarity Threshold (%): 95

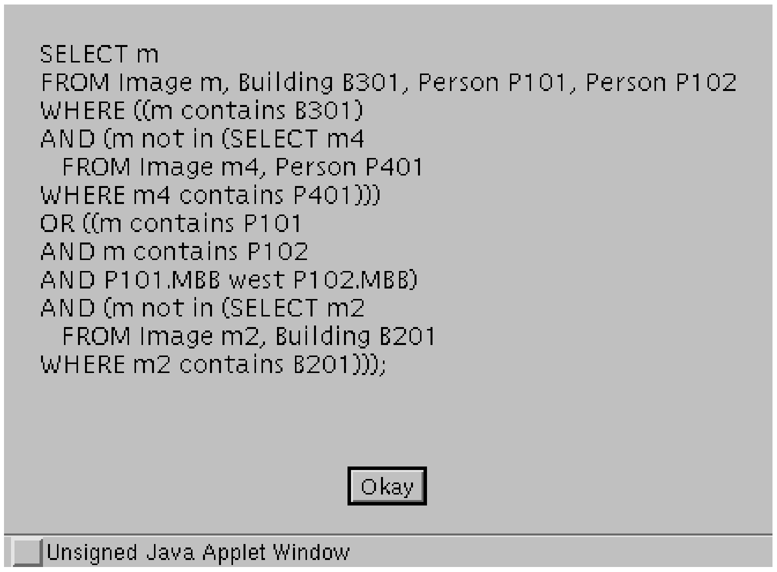
LSO

LSO Class	Attributes	Subclasses
Building	lastName	Scientist
Person	firstName	MovieStar
Animal	middleName	Politician
Vehicle	nationality	Athlete
	sex	
	yearOfBirth	

You are visitor number 573

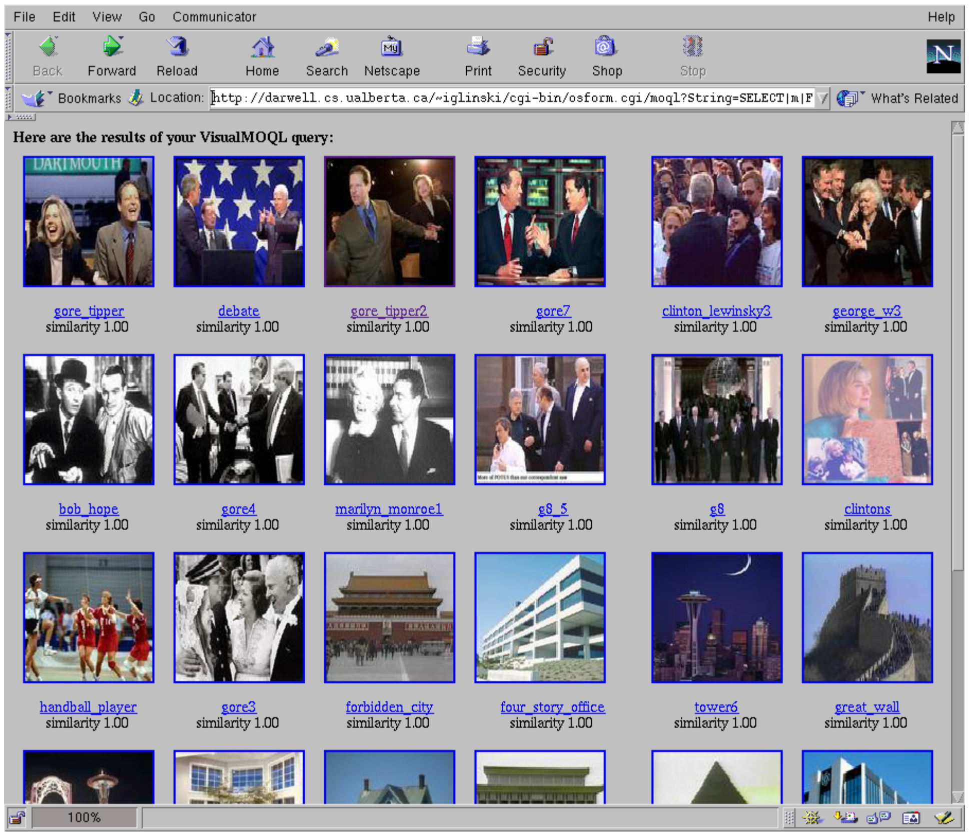
You can see the last 20 users

Figure 2.4: VisualMOQL Interface



```
SELECT m
FROM Image m, Building B301, Person P101, Person P102
WHERE ((m contains B301)
AND (m not in (SELECT m4
   FROM Image m4, Person P401
WHERE m4 contains P401)))
OR ((m contains P101
AND m contains P102
AND P101.MBB west P102.MBB)
AND (m not in (SELECT m2
   FROM Image m2, Building B201
WHERE m2 contains B201)));
```

Figure 2.5: Generated MOQL


File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://darwell.cs.ualberta.ca/~iglinski/cgi-bin/osform.cgi/moql?String=SELECT|m|F
What's Related
100%

Here are the results of your VisualMOQL query:

gore_tipper similarity 1.00	debate similarity 1.00	gore_tipper2 similarity 1.00	gore7 similarity 1.00	clinton_lewinsky3 similarity 1.00	george_w3 similarity 1.00
bob_hope similarity 1.00	gore4 similarity 1.00	marilyn_monroe1 similarity 1.00	g8_5 similarity 1.00	g8 similarity 1.00	clintons similarity 1.00
handball_player similarity 1.00	gore3 similarity 1.00	forbidden_city similarity 1.00	four_story_office similarity 1.00	tower6 similarity 1.00	great_wall similarity 1.00

Figure 2.6: Query Results

Chapter 3

Distributed Object Oriented Computing Platform

This chapter deals only with the OMA, a distributed object oriented computing platform provided by the Object Management Group. OMA is one of the infrastructure platforms that is used in the distributed DISIMA implementation. Its use in distributed DISIMA is discussed in Chapter 5.

3.1 Overview of OMA [27]

The Object Management Group (OMG) was formed in 1989 with the aim of developing interoperable, reusable, and portable distributed applications for heterogeneous systems, based on standard object oriented interfaces. The OMG solved this problem by introducing an architectural framework with supporting detailed interface specifications. One of the key industry standards produced by the OMG is the Object Management Architecture (OMA) and its core, the *Common Object Request Broker Architecture (CORBA)* specification. These provide a complete architectural framework that is both rich enough, and flexible enough, to accommodate a wide variety of distributed systems.

There are two related models in the OMA — the *Object Model* and the *Reference Model* — which describe how distributed objects and the communications among them can be specified in platform-independent ways. The Object Model provides an organized description of objects distributed over a heterogeneous environment, while the Reference Model categorizes interactions among these distributed objects.

The Object Model defines an object as an identifiable, encapsulated entity that provides services, through well-defined encapsulating interfaces, to clients, which are any entities capable of issuing requests to the object. The detailed implementations of services are transparent to clients. The Object Model describes not only basic object concepts, such as object creation and identity, requests and operations, and types and signatures, but also concepts related to object implementations, including methods, execution engines, and activation.

The Reference Model identifies and categorizes the components, interfaces and protocols that constitute the OMA. As Figure 3.1 shows, all four categories (*Object Services, Common Facilities, Domain Interfaces and Application Interfaces*) of object interfaces are conceptually linked by an *Object Request Broker (ORB)* component. This component enables communication between clients and objects, transparently activating those objects that are not running when requests are delivered to them, in a distributed environment:

- The *ORB* component of the OMA is the communications heart of the architecture, whose programming interfaces are defined in the CORBA specification [37] (see Section 3.2 for detailed discussion). It provides an infrastructure, allowing objects to communicate transparently, independent of the specific platforms and techniques used to implement the addressed objects. The ORB itself also provides an interface that can be used directly by clients, as well as objects. The ORB component will guarantee portability and interoperability of objects over a network of heterogeneous systems.

  The OMG *Interface Definition Language (IDL)* (see Section 3.2.2) provides a standard way to define CORBA objects' interfaces, in order to facilitate the interactions with objects and clients. The IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable the implementations of objects and sent requests to be programmed in any of the popular programming languages (Ada, C, C++, COBOL, Java or Smalltalk) at the developer's choice.

- *Object Services* are application domain independent, general purpose services that are essential for developing CORBA-based interoperable applications composed of distributed objects across multi-platform environments. The Object Services component provides standardized interfaces through the

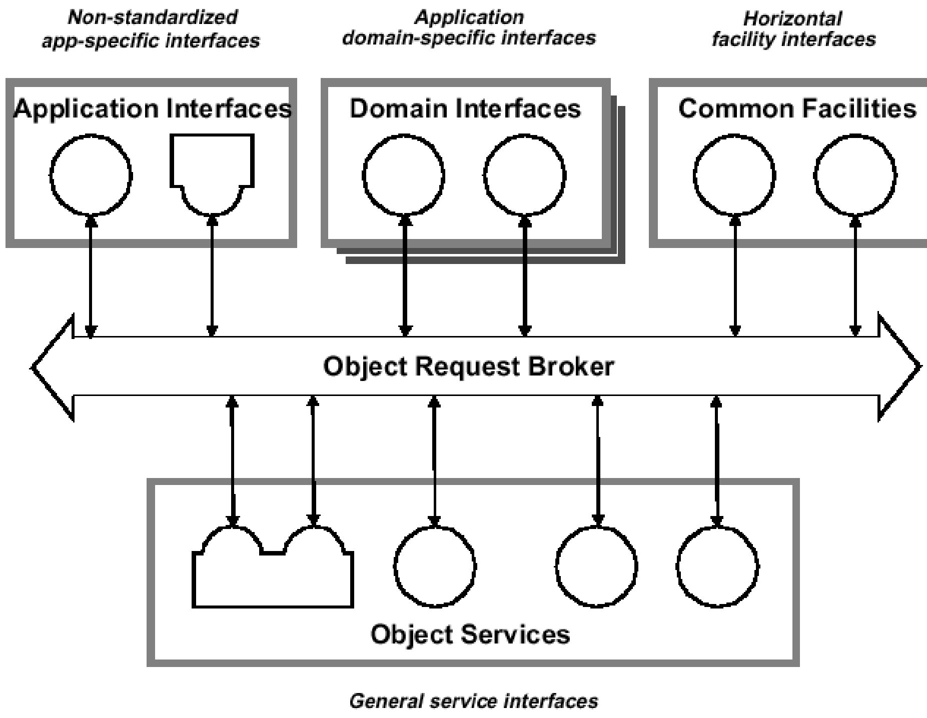

Non-standardized app-specific interfaces
Application domain-specific interfaces
Horizontal facility interfaces
Application Interfaces
Domain Interfaces
Common Facilities
Object Request Broker
Object Services
General service interfaces

Figure 3.1: OMA Reference Model: Interface Categories (Taken from [27])

OMG IDL to manage the life cycle of objects. This includes creating objects, controlling access to objects, keeping track of relocated objects, and consistently maintaining the relationship between groups of objects. Adopted Object Services used as standards by the OMG — including Naming, Events, Life Cycle, Persistent Object, Transactions, Concurrency Control, Relationships, Externalization, Licensing, Query, Properties, Security, Time, Collections, and Trading — are collectively called *CORBAservices* [30]. The Object Services supply operations that are used as building bricks by the other three interface categories, and are usually considered part of the core distributed object computing infrastructure. We will discuss some major object services in Section 3.3.

- *Common Facilities* are interfaces for *horizontal* end-user-oriented facilities, applicable to most application domains. The common facilities are used by many or most applications, regardless of application content. There are four major collections of these common facilities that are described in the OMG's specification, *CORBAfacilities* [26]: User Interface, Information Management, System Management, and Task Management. The Common Facilities provide higher level interoperable interfaces to objects from both Domain Interface and Application Interface categories. The operations provided by the Common Facilities are made available through the OMG IDL or through proposed extensions to OMG IDL.

- *Domain Interfaces* are application domain-specific interfaces, previously known as *Vertical* Common Facilities — such as Finance, Healthcare, Manufacturing, Telecommunication, Electronic Commerce, and Transportation. Industry groups are responsible for developing particular domain applications and the OMG will aid them in integrating their contributions into the OMA. As shown in Figure 3.1, there is a possible set of collections of Domain Interfaces grouped by application domains.

- *Application Interfaces* are non-OMG standardized application interfaces that are developed specifically in order for a given application to participate in the OMA. It is important to know that these applications themselves are not necessarily constructed using the object oriented pattern. Non-object oriented applications can be "wrapped" in objects, a subject which will be discussed in Section 3.2.

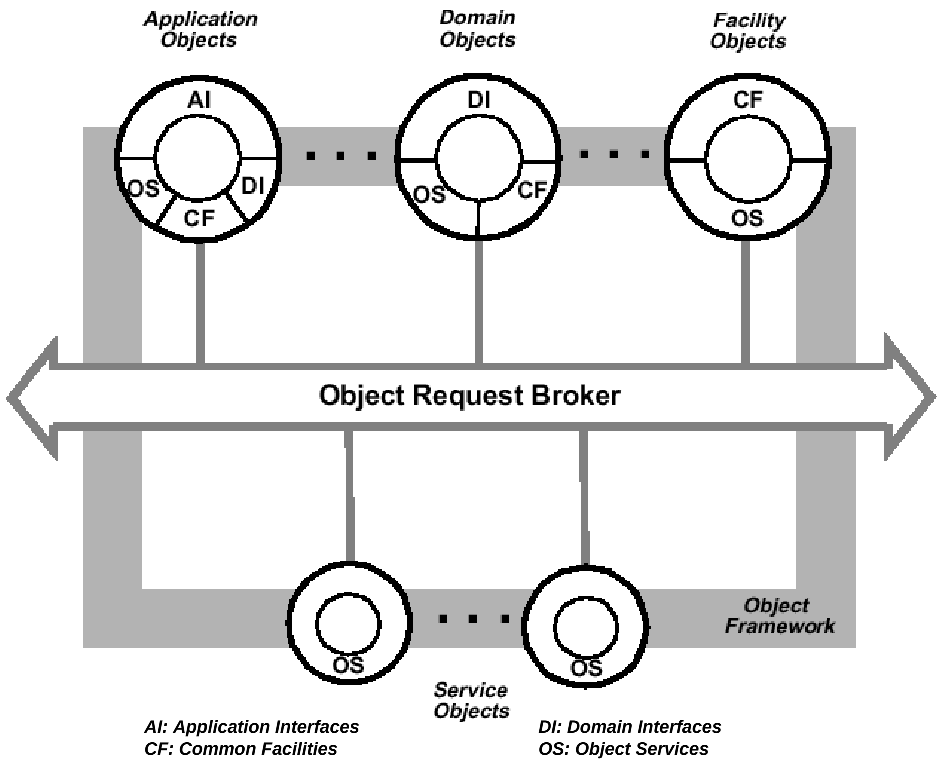


Figure 3.2: OMA Reference Model: Interface Usage (Adapted from [27])

Based on interface categories of the Reference Model, the OMG introduces another part of the Reference Model called domain-specific *Object Frameworks*, that focuses on interface usage, as illustrated in Figure 3.2. An Object Framework component is a group of collaborating objects that supply an integrated solution within an application or technology domain, and which can be customized by end-users. These objects are categorized into *Application, Domain, Facility,* and *Service Objects*, each of which supports or utilizes some combination of Application, Domain, Facility, and Service Objects interfaces. Notice that the objects shown in the figure are composed of two parts: an implementation core, and a concentric shell representing the interfaces that the object supports. Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in specific application or technology domains.

There are currently many commercial CORBA-compliant ORBs (e.g., BEA's ObjectBroker [3], Expersoft's PowerBroker [13], HP's ORB Plus [8], IBM's Component Broker [15], Iona's Orbix [43] and Sun's NEO [41]), as well as non-commercial CORBA-compliant ORBs (e.g., MICO [44], which will be discussed in Section 3.5), that can be chosen as a distribution and interoperability platform.

The latest version of CORBA specification is CORBA 3, but available ORBs coming from vendors only support version 2.3, which is the most mature specification.[1] As with all specifications adopted by OMG, CORBAservices and CORBAfacilities are defined only in terms of interfaces and their semantics, not a particular implementation. Only parts of CORBAservices have been implemented by vendors, and not much work on CORBAfacilities has yet been done.

3.2 CORBA

3.2.1 Overview

Figure 3.3 shows the structure of CORBA [21, 37] and the relationships among its components, which we will describe in detail in the following sections. The interfaces to the ORB are shown by shaded boxes, and the arrows indicate whether the ORB is called, or performs an up-call through the interface.

[1]By default, we refer to CORBA 2.3 as CORBA throughout the technical report, but in Section 3.4 we will also mention new features coming with CORBA 3.

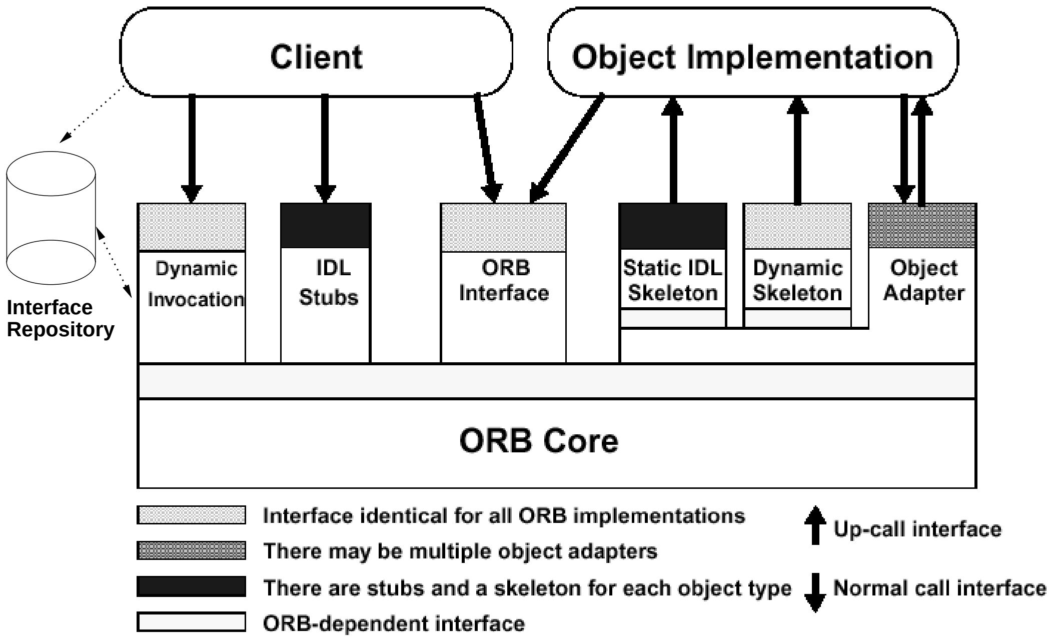


Figure 3.3: The Structure of Object Request Interfaces (Adapted from [37])

Basic Concepts and Terminology

We first define some important concepts and terms that are used in CORBA. Understanding these terms and concepts is critical to achieving a good understanding of CORBA itself.

- A *Client* is an entity that invokes a request/operation on an object by accessing its object reference. The client only knows the logical structure of the object according to its interface, and has no knowledge of the implementation of the object, where the implementation is located, or any aspect which is not reflected in the interface. The term “Client” exists only within the context of a particular request because the implementation of one object, itself may be a client of other objects.

- An *Object* refers to an abstract CORBA object that can be located by an ORB, i.e., a programming entity, which includes an identity (encapsulated in the object’s reference), an interface, and an implementation. The object’s components are transparent to the client, which invokes an operation on it.

- An *Object Reference* includes all the information used to uniquely specify a particular object within an ORB context. To both clients and object implementations, object references are opaque entities, whose actual representations are implemented only by the ORB. A client can use an object reference to invoke a request on an object, but it cannot create or modify an object reference.

- An *Object Implementation* actually implements an object by providing the code for the object’s method, and the data for the object instance. One implementation often includes other object implementations, or additional software, to achieve the behavior of the object. Normally, object implementations are independent of the ORB and the methods that clients invoke requests.

- A *Server* is a computational context in which the implementations of one or more objects exist. A server usually corresponds to a process. As with clients, this term is meaningful only within the context of a particular request. A given object could play both client and server roles.

The Common Object Request Broker Architecture

In the architecture, the ORB is responsible for all of the mechanisms required to catch the request from the client, to dispatch the request to the object implementation (server), to help the server process the request, and finally to send the result data, if applicable, back to the client. The *ORB Core* is an essential part of the ORB that provides the basic representation of objects and communication of requests.

The *Client* can use either an OMG *IDL stub* — the static stub depending on the interface of the target object — or the *Dynamic Invocation* interface (DII) — the interface independent of the target object's interface — to make a request. The Client can also directly communicate with the ORB core for some functions through the *ORB interface*.

As an up-call, the *Object Implementation* receives a request either through the OMG IDL-generated skeleton interface, the *Static IDL Skeleton* interface, or a *Dynamic Skeleton* interface (DSI) — the server side's analogue to the client side's DII. The Object implementation may call the *Object Adapter* (see Section 3.2.3) and the ORB core while processing a request, or at other times.

As we can see from Figure 3.3, there are two ways to define the interfaces to the objects.

- Interfaces are defined statically in the OMG IDL, which specifies the types of objects and the parameters to the operations. By this static approach, OMG IDL is mapped into programming language-specific stubs and skeletons that are compiled into the client program and the server program, respectively. Thus, a program (either a client or a server) can understand the types and functions of remote objects described in the OMG IDL. Through a stub, a client-side routine, a request can be invoked via a normal local function call in its programming language. Similarly, a skeleton is a server-side function that allows a request received by a server to be dispatched to the appropriate object implementation.

- Sometimes, interfaces to objects cannot be decided at compile time. CORBA provides an *Interface Repository (IR)* into which the IDL information of these objects can be added at run time. The IR is a service that represents the components of an interface as objects, permitting run-time access to these components, and keeps additional information associated with interfaces to ORB objects such as debugging information, and libraries of stubs or skeletons (see Figure 3.3).

Although no compile time interface information is available, DII and DSI allow dynamic construction and handling of requests at run time, rather than at compile time, by accessing the IR service. Instead of calling a specific stub routine to a particular operation on the object, a client can create a request by specifying the object to be invoked, the operation to be performed, and the set of arguments for the operation through the DII.[2] Rather than being invoked through a specific skeleton to a particular operation, an object implementation is located through the DSI, that provides access to the operation name and arguments in a way similar to the client side's DII [37].

The static way provides a more natural programming model while the dynamic approach is especially useful for situations in which a client does not know the type or interface of the target object at compile time, and yet is able to invoke a valid request to that object at run time.

According to CORBA's specification, clients can invoke requests in the following three communication modes:

Synchronous Mode: In this mode, a client blocks while it waits for the completion of the request. Obviously, it is quite inefficient since those requests can be processed parallelly by multiple objects.

Deferred Synchronous Mode: After a client invokes a request, it continues executing, and keeps polling to get the results until the operation is completed, or the client purposely stops polling. The client can invoke the request in this mode by using only the DII.

One-way Mode: Invoked in this mode, a request performed by the client is not guaranteed to be delivered to the target object, and the client is not allowed to get any results from the operation.

[2]In particular, creating a DII request cause the ORB to transparently access the IR to obtain information about the types of the arguments and return values.

Communication Modes	Static Invocation	Dynamic Invocation
Synchronous	Yes	Yes
Deferred Synchronous	No	Yes
One-way	Yes	Yes

Table 3.1: Invocation Types and Communication Modes

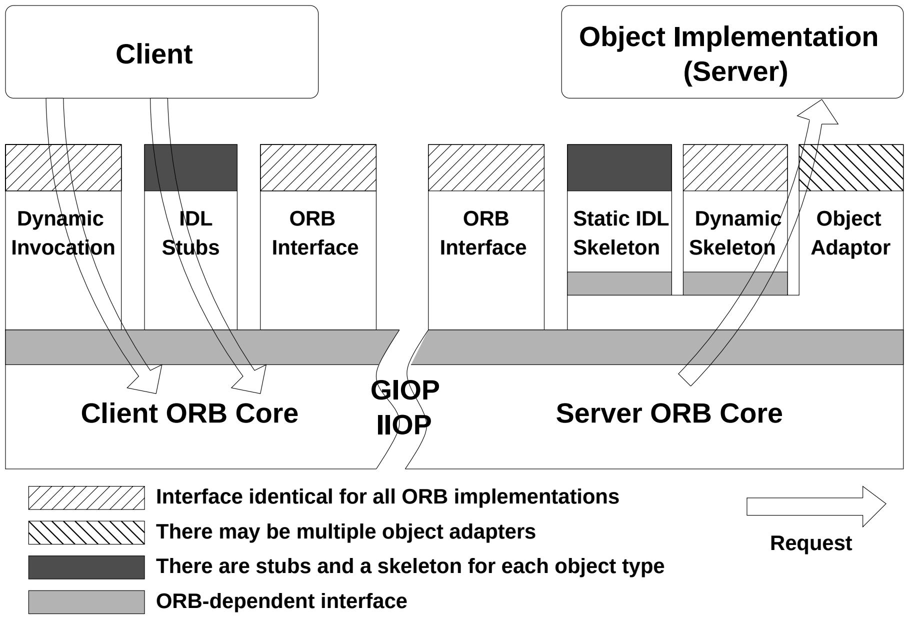


Figure 3.4: Invoking and Dispatching Requests (Adapted from [37])

Current CORBA-compliant commercial ORBs do not support the *asynchronous* mode of requests[3], but asynchronism can be simulated by making two objects send one-way requests to each other.

Table 3.1 shows the relationship between invocation types and communication modes.

Figure 3.4 shows how requests from the Client flow down through the ORB and up into the Object Implementation (the Server) in the following steps:

[3]Although CORBA 3 does support the asynchronous mode — which is discussed in Section 3.4 — no CORBA 3 compliant commercial ORBs are available as of time of this writing.

1. A client initiates a request through an object reference by calling stub procedures (the static stub interface) for that particular object, or by constructing the request dynamically (the DII), knowing exactly the type of the object and the desired input/output arguments.

   Either way, the request is sent to the client ORB core, and the target object cannot tell in which way the request was invoked.

2. If the client ORB core cannot locate the target object for this request, it transmits the request to the ORB core linked with the target object implementation (the server) by *General Inter-ORB Protocol (GIOP)/Internet Inter-ORB Protocol (IIOP)* (see Section 3.2.4).

3. The server ORB core dispatches the request to the object adapter that created the target object.

4. The object adapter locates the appropriate object implementation, transmits arguments, and transfers control to the object implementation. Like the client, the server can choose either a static (static IDL skeleton interface) or dynamic (DSI) dispatching mechanism for its object implementation.

   While processing the request, the object implementation may obtain some services from the ORB through the object adapter, or from the ORB core directly.

5. After the object implementation finishes the request, it returns the control and the output values back to the client.

The following sections describe other important CORBA components.

3.2.2 OMG IDL

The OMG IDL defines the types of objects by specifying their interfaces. An object's interface is composed of a set of client-accessible operations, and input/output data to those operations. The OMG IDL is purely a declarative language and therefore clients and servers cannot be implemented directly in IDL. The only purpose of the OMG IDL is to make interfaces to be defined in a language independent way, which allows applications developed in different programming language to interoperate.

To use or implement an interface, the language-independent interface defined in the OMG IDL must be mapped (using the IDL compiler) into corresponding

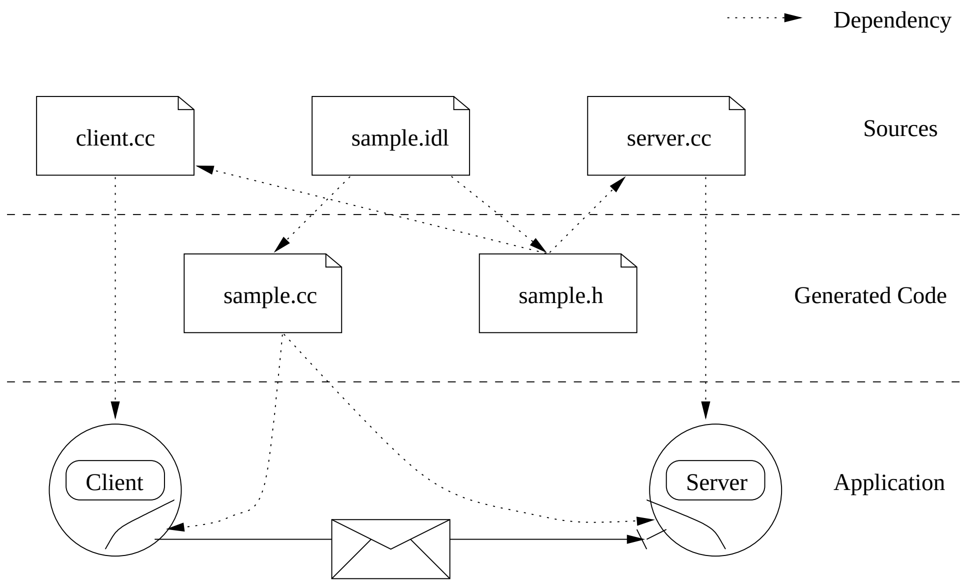


Figure 3.5: The OMG IDL Mappings (Taken from [44])

types definitions and APIs of a particular programming language. These types and APIs are used by the developer to provide application functionality and to interact with the ORB.

Figure 3.5 shows the creation process of a client and a server in C++, as implemented in the MICO ORB [44]. After compiling the source IDL file *sample.idl*, the IDL compiler will generate two files: *sample.h* and *sample.cc*. The former contains class declarations for the base class of the *sample* object implementation and the stub class that a client will use to invoke methods on remote *sample* objects. The latter contains implementations of those classes and some supporting codes. The client side file *client.cc* includes ORB initialization and invocation of the method on a particular interface declared in the IDL file. The files *client.cc* and *sample.h* are compiled to create an objective file *client.o*, using any C++ compiler. The file *sample.cc* is compiled to an objective file *sample.o* which is linked with *client.o* to create an executable CORBA client. The server side file *server.cc* includes the actual implementation codes for the methods. Similarly, the C++ compiler uses the files *server.cc*, *sample.h*, and *sample.cc* together to create an executable CORBA server.

The OMG IDL obeys the same lexical rules as C++, but adds some new key-

abstract	double	long	readonly	unsigned
any	enum	module	sequence	union
attribute	exception	native	short	ValueBase
boolean	factory	Object	string	valuetype
case	FALSE	octet	struct	void
char	fixed	oneway	supports	wchar
const	float	out	switch	wstring
context	in	private	TRUE	
custom	inout	public	truncatable	
default	interface	raises	typedef	

Table 3.2: The OMG IDL Keywords (Taken from [37])

words (e.g., *any, attribute, interface, module, sequence,* and *oneway*) to support distribution concepts. Table 3.2 lists all the keywords supported in the OMG IDL.

The IDL grammar is a subset of the adapted ANSI C++ standard that supports syntax only for constant, type, and operation declaration, and not for any algorithmic structures or variables. It includes additional constructs to support the operation invocation mechanism. As specified in CORBA, The OMG IDL grammar is more restrictive than the C++ syntax in the following ways:

- A function return type is mandatory.
- A name must be supplied with each formal parameter to an operation declaration.
- A parameter list consisting of the single token *void* is not permitted as a synonym for an empty parameter list.
- Tags are required for structure, discriminated unions, and enumerations.
- Integer types cannot be defined as simply int or unsigned; they must be declared explicitly as *short, long*, or *long long*.
- *char* cannot be qualified by *signed* or *unsigned* keywords.

An important feature of the OMG IDL interfaces is that they can inherit from one or more other interfaces. This feature allows new interfaces to be defined by existing ones and, since a derived interface inherits all attributes and operations defined in all of its base interfaces, objects implementing a new derived interface can be substituted anywhere objects supporting the base interfaces are allowed (the well-known substitutability concept in object-oriented programming). Consider the second example below. The *AccountTransaction* interface is derived from the *Account* interface. Anything dealing with objects of type *Account*, can also use an object supporting the *AccountTransaction* interface because such an object also supports the *Account* interface.

Following are two OMG IDL examples. The first defines the interface through which the client can get sets of images from an image DBMS. This example focuses on complicated user defined types:

```
//Example 1 - Filename ‘‘queryagent.idl’’

module DB {

  interface QueryAgent {

    //exception
    exception SyntaxError {
      unsigned short position;
      string errMessage;
    };

    //user defined types
    typedef sequence<octet> streamType;

    //content of an image
    struct imageType {
      string serverId;
      string imageId;
      string imageLabel;
      streamType imageStream;
      float similarity;
    };

    typedef sequence<imageType> imagesType;

    //operation
28     imagesType getImages (in string queryString)
29                    raises (SyntaxError);
30
31   };//end of interface QueryAgent
32
33 };//end of module DB
```

IDL uses the *module* construct to create namespace, preventing pollution of the global namespace. Line 3 defines the module `DB`. The declaration of the interface `QueryAgent` starts at line 5. Lines 8-11 define the content of the exception this interface may raise. Line 14 defines the type for the image stream, which is an unbounded octet array. Lines 17-23 define the structure of the image type, which includes the server Id (indicating which database this image comes from), the image Id (an octet sequence that uniquely identifies the image in the database where it comes from), the image label (name), the image stream (real image data), and the similarity between the target image and the query image. Line 25 defines the return type of the query, i.e., a set of images. Lines 28-29 declare an operation (method) called `getImages`. The input parameter is query string that will be passed along to the image database; the output is a set of result images matching the query.

The second example below demostrates interfaces to bank account transactions. Most features of the IDL are covered, although some operations are not realistic. This example also demonstrates the substitutability principle discussed earlier. The *AccountTransaction* interface is derived from the *Account* interface. Anything dealing with objects of type *Account* can also use an object supporting the *AccountTransaction* interface, because such an object also supports the *Account* interface.

```
//Example 2 - Filename "bank.idl"

//establish a unique prefix for interface repository Ids
#pragma prefix "bank.com"

module BANK {

  //types
  enum AccountType {CHECKING, SAVING};
  //constant
  const unsigned long MAX_LENGTH = 20;
```

```
typedef sequence<char, MAX_LENGTH> AccountNum;

interface Account {
  //attributes
  readonly attribute AccountNum check_account_num;
  readonly attribute float check_account_balance;
  readonly attribute AccountNum save_account_num;
  readonly attribute float save_account_balance;
  readonly attribute string pin;
           attribute string address;
};//end of interface Account

//interface inheritance:
//AccountTransaction inherits attributes of Account
interface AccountTransaction : Account {

  //exceptions
  exception account_invalid {
    string reason;
  };
  exception incorrect_pin {};

  //operations and raising exceptions
  float balance (in AccountType account_type,
                 in AccountNum account_num,
                 in string pin)
         raises (account_invalid, incorrect_pin);

  void deposit (in AccountType account_type,
                in AccountNum account_num,
                in float amount,
                out float new_balance)
        raises (account_invalid);

  //one-way
  oneway void withdraw (in AccountType account_type,
                        in AccountNum account_num,
                        in float amout,
                        in string pin);

52  };// end of interface AccountTransation
53
54 };// end of module BANK
```

CORBA provides an IR that allows run-time access to the IDL definition. The IDL compiler assigns a repository Id as a unique name for each IDL type into the IR. The prefix "pragma" adds a unique prefix to a repository Id to ensure its uniqueness. Line 4 defines a prefix `bank.com`. Line 9 declares an enumerated variable `AccountType` with `CHECKING` and `SAVING` representing types of a bank account. Lines 11-12 define the type of an account number which is a bounded sequence with maximum length of 20. Lines 14-22 define an interface *Account* which includes the primary information of a bank account, such as pin number, home address, account numbers, account balances. An *attribute* defines read and write operations on a variable. A *readonly* attribute defines a single read operation on a variable. Line 20 is semantically equivalent to the preceding codes:

```
  string get_pin ();
```

Line 21 is semantically equivalent to the following codes:

```
  string get_address();
  void set_address (in string address);
```

Even though attribute definitions look like variables, in fact they are just shorthand for definition of a pair of operations (or a single operation for *readonly*). Lines 26-52 define an interface `AccountTransaction`, including some basic transaction related to a bank account (e.g., chequing balance, deposit, withdraw). Lines 29-32 define the contents of exception. Lines 35-38 define a method `balance`, and lines 40-44 define a method `deposit`. Lines 47-50 define the *oneway* operation `withdraw`. Note that the oneway operation cannot return any values, nor can it raise exceptions.

3.2.3 Object Adapters

An *Object Adapter* [21, 39] is the primary means through which an object implementation accesses most services provided by the ORB, such as:

- Generation and interpretation of object references
- Method invocation
- Security of interactions

- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations
- Registration of implementations

An object adapter has a public interface to the object implementation, and two private interfaces — one to the skeleton, and the other to the ORB core. The intention is to isolate object implementation from the ORB core as much as possible.

There is a variety of possible object adapters, according to different requirements of object implementations. Most object adapters are designed to cover a range of object implementations.

Prior to CORBA 2.1, the *Basic Object Adapter (BOA)* was the only CORBA object adapter, and provided a minimum of functionality to object implementations. As a consequence, many ORB vendors added custom extensions to BOA to support more complex operations upon object adapters, resulting in poor compatibility of object implementations among different ORB vendors.

In CORBA 2.2, A new object adapter called *Portable Object Adapter (POA)* was introduced to replace the BOA. The POA provides more extended interfaces than the BOA, and fulfills the needs of most object implementations. Therefore, the BOA specification has been removed from CORBA.

POA Related Concepts and Terminology

- A *Servant* is a programming language object or entity that implements requests on one or more objects, providing bodies or implementations for those objects. Servants generally exist within the context of a server process. A request made on an object through its reference is interpreted by the ORB and transformed into an invocation on a specific servant.
- An *Object Id* is an octet sequence that uniquely identifies a particular abstract object within the scope of its host POA. Object Id values may be assigned and managed by the POA, or by the user-supplied implementation. Encapsulated by object references, Object Id values are hidden from clients.

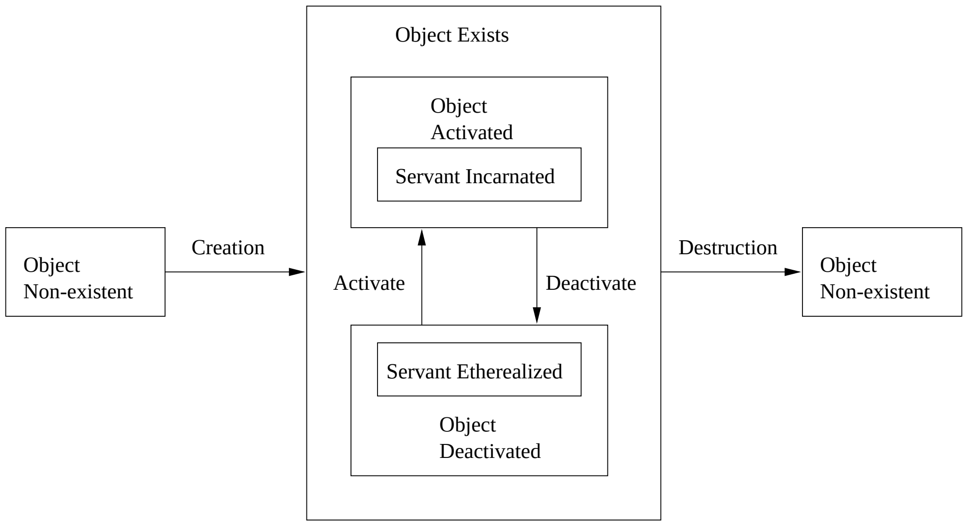


Figure 3.6: Object and Servant Life Cycles (Taken from [21])

- An *Active Object Map* is a table maintained by an object adapter that maps its active objects to their associated servants. Active objects are identified in the map via Object Ids.
- A *Policy* is an object associated with a POA by an application in order to specify a characteristic shared by the objects implemented in that POA.

POA is responsible for the entire life cycle of a CORBA object – from its creation to its destruction. Figure 3.6 shows the life cycle states of objects with respect to the life cycle of their servants. When a request is received by the POA (via the ORB) for the invocation of an object, the POA *creates* the CORBA object that will service that request. Creation of an object associates a servant with it. A created object is *activated* by its servant. The object must be *incarnated* by a servant to have requests delivered to it. When the servant is finished with the CORBA object, the servant is *etherealized* and its linkage to the object is broken [21]. The object is then *deactivated*. A created object can alternate between the active and deactive modes during its life-cycle. Eventually, the object is *destroyed*, which completes its life-cycle.

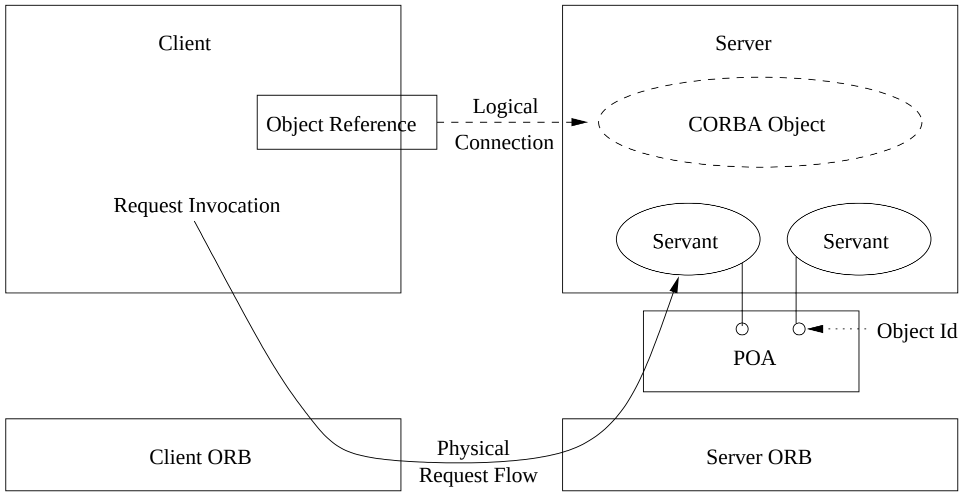


Figure 3.7: Abstract POA Model (Adapted from [21])

Abstract POA Model

Figure 3.7 shows the abstract POA model while a request sent from the client is dispatching to the servant. First, the server exports an object reference for an object. By accessing the object reference, perhaps via the *Naming Service* or the *Trading Service* (see Section 3.3), the client invokes a request. The client ORB uses the object reference to dispatch the request to the server ORB. The server ORB then dispatches the request to the POA hosting the target object and, finally, the POA further dispatches the request to the appropriate servant (identified by the Object Id) that incarnates the target object. In Figure 3.7, the straight arrow between the object reference and the object indicates the logical connection between them, while the curved arrow represents the physical request flow.

POA Policies

A major feature of the POA is that a server application can have more than one POA, each of which represents a set of objects that have similar properties. These properties are controlled via POA *policies* that are specified when a POA is created. Each server application has at least one POA called *Root POA*, which stores a standard set of policies. (In the following description of POA policies, the policy values ending with '*' are default policy values for Root POA.) A nested POA can be created on an existing POA, from which the new POA inherits policy values by default. The POA policies are described below:

- *Thread Policy:* A POA can either have the ORB control its threads (`ORB_CTRL_MODEL*`) or be single-thread (`SINGLE_THREAD_MODEL`). If single-thread, all requests are processed sequentially. Even though in a multi-threaded environment, all requests are synchronized to only execute one request at a time. In contrast, if ORB-controlled threading policy is specified, the ORB controls the use of threads.

- *Lifespan Policy:* This policy is to specify whether the objects created within a POA are transient (`TRANSIENT*`) or persistent (`PERSISTENT`). Transient objects are CORBA objects whose lifetimes are bounded by the lifetimes of the server processes in which they are created, while the lifetimes of persistent objects are independent of those of any server processes in which they are activated.

- *Object Id Uniqueness Policy:* This policy allows the server to control whether a servant can be associated with only a single object (`UNIQUE_ID*`), or with multiple objects (`MULTIPL_ID`). The servants associated with multiple objects can reduce the server's memory use.

- *Id Assignment Policy:* The `SYSTEM_ID*` policy means that objects created with a POA are assigned Object Ids only by that POA; otherwise (`USER_ID`), Objects Ids are assigned only by the server.

- *Servant Retention Policy:* This policy decides whether a POA retains (`RETAIN*`) the associations between servants and objects, or whether it establishes a new association for each incoming request. The `NON_RETAIN` policy requires either `USE_DEFAULT_SERVANT` or `USE_SERVANT_MANAGER` policies.

- *Request Processing Policy:* When a request arrives for a specific object, the POA can act as follows:
  - `USE_ACTIVE_OBJECT_MAP_ONLY*` – If the Object Id is not found in the Active Object Map, an `OBJECT_NOT_EXIST` exception is returned to the client. The `RETAIN` policy is also required.

  - `USE_DEFAULT_SERVANT` – If the Object Id is not found in the Active Object Map or the `NON_RETAIN` policy is present, and a default servant has been registered with the POA using the *set_servant* operation, a default servant can be registered. The `MULTIPLE_ID` policy is also required.
  - `USE_SERVANT_MANAGER` – If a *servant manager* has been registered with the POA, it is invoked by the POA to locate a servant or raise an exception.

  The combination of these policies with the `RETAIN` policy provides flexibility to control servant registration and allocation within the server process.

- *Implicit Activation Policy:* If the implicit activation (`IMPLICIT_ACTIVATION*`) policy is chosen, a POA can activate a servant implicitly. This is useful for registering servants for transient objects. For instance, a server can create a servant by instantiating the server class and, invoking its $\_this$ method, it registers the servant and creates an object reference for the object, in one single operation, in C++ language. `IMPLICIT_ACTIVATION` also requires the `SYSTEM_ID` and `RETAIN` policies. The other value of this policy is `NO_IMPLICIT_ACTIVATION`.

3.2.4 ORB Interoperability

ORB interoperability [50] specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs [37].

As specified in CORBA, the elements of interoperability include:

- ORB interoperability architecture,
- Inter-ORB bridge support, and
- General and Internet inter-ORB Protocol (GIOPs and IIOPs)

In addition, the architecture accommodates *Environment-Specific Inter-ORB Protocols (ESIOPs)* that are optimized for particular environments.

The ORB Interoperability Architecture provides a conceptual framework for defining the elements of interoperability and for identifying its compliant points. Specifically, the architecture initiates the concepts of *immediate* and *mediated* bridging between ORB domains. The IIOP forms the common basis for both immediate and mediated bridgings.

The architecture clearly identifies the roles of different kinds of domains for ORB-specific information. ORBs in the same domain can communicate directly, while communication of ORBs in different domains must be achieved by a bridge that fully maps the content and semantics of one ORB to that of the other.

The GIOP is a very basic protocol built for ORB-to-ORB communications, and it is designed to be simple, scalable, and easy to implement. The IIOP, which is the basic inter-ORB protocol for Transmission Control Protocol/Internet Protocol (TCP/IP) environments, is one of the variants of the GIOP and shares the GIOP's features. The IIOP is a mandatory protocol for inter-bridge communications.

3.3 CORBAservices: Common Object Services

As defined by OMG, Object Services are "*interfaces and sequencing semantics that are widely available and are mostly commonly used to support building well-formed applications in a distributed object environment built on a CORBA-compliant ORB.*" [30] Among all 15 object services described in the CORBAservices specification ([30]), *Naming Service, Event Service, Life Cycle Service,* and *Persistent Object Service* are the most fundamental object services in the OMA, and were the first adopted by the OMG as industry standards. During the subsequent couple of years, more and more object services (including *Transaction Service, Concurrency Control Service, Relationship Service, Externalization Service, Licensing Service, Query Service, Property Service, Security Service, Time Service, Collections Service,* and *Trading Service*) were proposed by OMG members, and added to the Object Services interface category — which provides a better foundation for other interface categories in the OMA. Some of these are important database related object services, including *Transaction Service, Backup and Recovery Service, Concurrency Control Service, and Query Service*. As we mentioned before, only some of these Object Services are actually implemented by the vendors. The following sections give a brief description of some major object services.

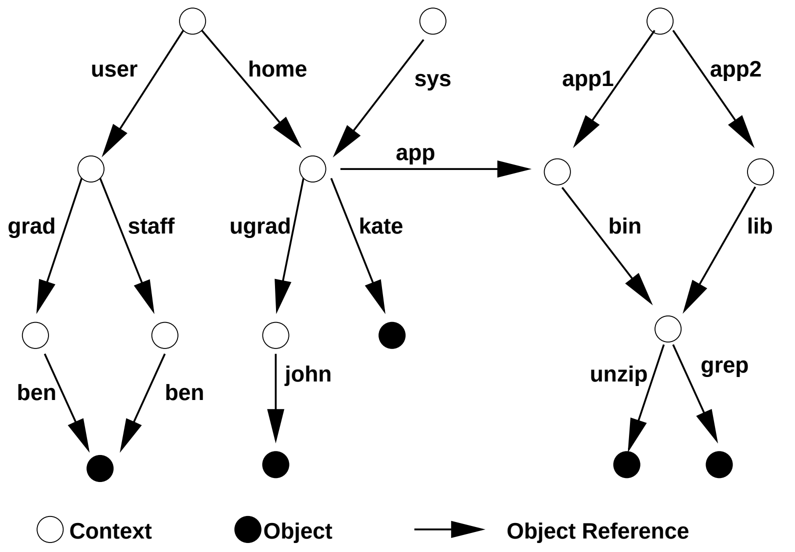


Figure 3.8: A Naming Graph

3.3.1 Naming Service

The Naming Service supports a name-to-object association called a *name binding*, i.e., to bind a name to an object relative to a *naming context*. A naming context is an object that contains a set of name bindings where each name is unique. Different names can be bound to an object in the same or different contexts at the same time, but each name can only identify exactly one object. To resolve a name is to determine the object associated with the name in a given context.

A context is like any other object, and it can be bound to an object or another context object. It forms a hierarchy of contexts and bindings known as a *naming graph*, which can be supported in a distributed, federated fashion. A naming graph is like a file system, in which contexts are analogous to directories that store bindings either to directories (other contexts) or files (objects). See Figure 3.8.

3.3.2 Event Service

The Event Service supports asynchronous events by decoupling the communication between objects. It defines two roles for objects: the *supplier* role, which produces event data, and the *consumer* role, which processes event data. Event data are communicated between suppliers and consumers by issuing standard CORBA requests through appropriate event channel implementations.

The *push model* and the *pull model* are two approaches defined to initiate event communication. The push model allows a supplier of events to initiate the transfer of the event data to consumers, while the pull model allows a consumer of events to request the event data from a supplier. The communication can be generic, using a single parameter that packages all the event data, or typed, using operations defined in OMG IDL.

An *event channel* is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously, and is itself both a consumer and a supplier of events.

3.3.3 Trading Service

The Trading Services facilitates objects advertising their capabilities and matching their needs against advertised capabilities. A *trader* is an object that supports the trading object service in a distributed environment.

The *service provider* registers the availability of the service by invoking an *export* operation on the trader, giving the trader a description of a service, and the location of an interface where that service is available. To *import*, an object asks the trader for a service having certain properties against its needs. The trader checks the service description it holds, and gives the object reference of the selected service to the importer. Then, the importer is able to communicate with the service.

3.3.4 Life Cycle Service

The *Life Cycle Service* defines a framework composed of services and conventions for creating, deleting, copying, and moving objects. A *client* is any piece of code that initiates a life cycle operation for some object. Clients can perform life cycle operations on objects in different locations under the conventions of the Life Cycle Service.

A client model of creation is defined in terms of *factory* objects that provide the client with specialized operations to create and initialize new instances for the implementation. A factory has no standard interface, but a *generic factory* interface. Clients can delete, move, or copy an object by invoking remove, move, or copy requests, respectively, on *target* objects that support *LifeCycleObject* interfaces. The Life Cycle Service also defines *factory finder* objects which support a `find_factories` operation for returning a sequence of factories. Clients pass factory finders to the move and copy operations that invoke the `find_factories` operation, to find a factory to interact with. The new copy or the migrated object will be within the scope of the returned factory.

3.3.5 Persistent Object Service

The *Persistent Object Service* provides common interfaces to the mechanisms used for retaining and managing the persistent state of objects in a data storage-independent fashion. Objects can be considered in two states: the *dynamic state*, which is typically in memory and transient; and the *persistent state*, which is used to reconstruct the dynamic state. The Persistent Object Service is primarily responsible for storing the persistent state of objects.

Each object ultimately has the responsibility of managing its own state, but can use, or delegate to, the Persistent Object Service for the actual work. There is no requirement that any object use any particular persistence mechanism. The Persistent Object Service provides capabilities that support various styles of usage and implementation, in order to be useful to a wide variety of objects. The architecture of the Persistent Object Service has multiple components and interfaces. The interfaces allow different implementations of the components to work together to obtain different qualities of service.

3.3.6 Transaction Service

The *Transaction Service* supports concepts of transactions including *flat* transactions (mandatory in the specification), and *nested* transactions, which have the following properties, known as ACID [20]:

Atomicity - Either all of the actions of a transaction are committed, or none are. Thus, if a transaction is interrupted by failure, all efforts are undone (rolled back).

Consistency - A transaction maintains a consistent state.

Isolation - A transaction is isolated from other transactions. Its intermediate states are not visible to other transactions. Transactions appear to execute sequentially even though they are performed concurrently.

Durability - Once a transaction commits (completes), its efforts are persistent and survive future system failures.

The Transaction Service defines interfaces that allow multiple, distributed objects to cooperate in order to provide the above properties. Transaction semantics can be defined as part of any object that provides ACID properties. This service depends on the *Concurrency Control Service* to enforce isolation, and the *Persistent Object Service* to enforce durability.

3.3.7 Concurrency Control Service

The purpose of this service is to coordinates the concurrent access to a single shared resource (an object), such that the resource remains in a consistent state when accessed concurrently by multiple clients. The Concurrency Control Service ensures that *transactional* and *non-transactional* clients are serialized.

Coordination is achieved by preventing multiple clients from simultaneously processing *locks* (each lock is associated with a single resource and a single client) for the same resource in a conflicting mode. Different lock modes are defined to provide flexible conflict resolution.

The Concurrency Control Service, together with the *Transaction Service* can be used to coordinate the activities of concurrent transactions.

3.3.8 Query Service

The *Query Service* provides query operations on collections of objects — including predicate-based declarative specifications — and may return collections of objects. Queries can be specified by object derivatives of SQL and/or other styles of object query languages including direct manipulation query languages.

Queries are either executed on the source object collections by the application of predicates, or by intermediate collections that are produced by *query evaluators*. Query evaluators apply a given predicate to collections to generate other collections. They can operate on implicit collections of objects through their OMG IDL interfaces. Thus, the query service supports nested queries of traditional form.

3.3.9 Collections Service

The *Collections Service* provides a uniform way to create and manipulate the most common collections (groups of objects such as sets, queues, stacks, lists, binary trees) generically. Three categories of interfaces are defined to accomplish this purpose:

1. *Collection interface* and *collection factories*. A client uses a collection factory to create a collection instance of a chosen collection interface which offers grouping properties that match the client's requirements. A client uses collections to manipulate elements as a group.

2. *Iterator interfaces*. An iterator is created for a given collection, which is the factory for it. An iterator is used to traverse the collection in a user-defined manner, process elements it points to, mark ranges, etc.

3. *Function interfaces* A client creates user-defined specializations of these interfaces using user-defined factories. Instances of function interfaces are used by a collection implementation, rather than by a client.

3.3.10 Other Object Services

The *Externalization Service* describes protocols and conventions for object externalizing and internalizing. To externalize an object is to record the object's state in a stream of data (in memory, on a disk file, across the network, etc.). It can then be internalized into a new object using the same or a different process.

The *Relationship Service* allows entities, represented as CORBA objects, and relationships to be explicitly represented

The *Licensing Service* provides a mechanism for producers to control the use of their intellectual properties.

The *Property Service* is used to dynamically associate named values with objects outside the static IDL-type system.

The *Security Service* consists of the following features: Identification and authentication, authorization and access control, security auditing, security of communication, non-repudiation, and administration.

The *Time Service* enables the user to obtain current time, together with an error estimate associated with it, to synchronize clocks in distributed systems.

3.4 CORBA 3

In CORBA 3 [40], some important features are added into the current CORBA to increase its capability and ease-of-use. Although the final official OMG specifications have not yet come out, the draft specifications are now available. They are divided neatly into three major categories: *Internet Integration*, *Quality of Service Control*, and *The CORBAcomponent Architecture*.

3.4.1 Internet Integration

Firewall Specification

The *Firewall* Specification [29] provides specifications and descriptions of how to achieve inter-ORB interoperability through firewalls, and a bi-directional GIOP connection useful for callbacks and event notifications.

There are two types of firewall: transport-level and application-level. The specification currently supports TCP (transport-level), SOCKS (transport-level) and GIOP (application-level) firewalls. For each of these firewalls, the specification provides feasible solutions to cope with CORBA traffic over the IIOP protocol, including IIOP over *Secure Socket Layer (SSL)*.

In CORBA, objects often need to callback the client that invoked them. Because standard CORBA connections carry invocations asymmetrically, a callback typically requires the establishing of a second TCP connection for this traffic, in a reverse direction. The essential problem with callbacks is that the target host (where the client is) of callback operation invocation is usually a workstation rather than a server host (where objects are). Since a firewall does not allow any incoming TCP connection to an inside workstation, except to certain well-known and carefully configured hosts like HTTP or FTP servers, this callback technique is not acceptable across firewalls. The specification indicates that an IIOP connection can accomplish the callback functionality under certain restrictive conditions, that do not compromise security at either end of the connection.

Interoperable Name Service [31]

As we know, the object reference is a foundation of the whole CORBA architecture. A client cannot access a remote object implementation (even though it knows where the object implementation is located and that the object implementation is running) unless the client can get the object reference of the object. CORBA provides a Naming Service where the client can access the reference of the object implementation, but what if the reference of the Naming Service itself is not available? Suppose we know only that the Naming Service is running on a machine whose domain name is darwell.cs.ualberta.ca. The *Interoperable Name Service* supports the use of *Uniform Resource Locator (URL)*-based names, and defines two types of URL-format object references that are user friendly and similar to FTP and HTTP URLs: *iiopploc* provides stringified references that can reach defined services (like Naming Service) at a remote location represented by URLs, while *iiopname* allows URLs to denote entries in a Naming Service.

In the above example, we could use an iiopploc identifier:

$$iioploc : //darwell.cs.ualberta.ca/NameService$$

to get the Naming Service running on that machine.

3.4.2 Quality of Service Control

Asynchronous Messaging and Quality of Service Control [28]

Two new invocation modes: *Asynchronous Method Invocation (AMI)* and *Time Independent Invocation (TII)*, both of which can be used in static and dynamic invocations, are introduced in the *Messaging* Specification [28]. Clients can use either polling or callback methods to get the results of invocations. Upon receiving these invocations, *Routers* handle the passing of messages between clients and target objects, and communicate with them. Some policies are specified to control the *Quality of Service (QoS)* of asynchronous invocations.

Minimum [32], Fault-Tolerant [35], and Real-Time CORBA [36]

minimumCORBA refers to a subset of CORBA designed for some systems with limited resources, such as embedded systems. Once such systems are finalized, their communications with outside world are predictable. Therefore, the dynamic aspects of CORBA, such as DII and IR, are omitted — while portability, interoperability, and full IDL support are still the goals of minimumCORBA.

Real-time CORBA consists of some optional extensions added to CORBA to be used in a real-time system. A real-time system has special requirements on both resources management and predictability of system execution. Real-time CORBA not only helps developers meet the requirements of real-time systems, but also brings them the same benefits of implementation flexibility, portability, and interoperability which CORBA provides.

Fault Tolerant CORBA provides robust support for high level reliability applications which require absolutely no failure. Its standard is based on entity redundancy (e.g., replication of objects), fault detection, and recovery management.

3.4.3 The CORBAcomponent Architecture

CORBAcomponents [34] and CORBAscripting [33]

CORBAcomponents introduces a new meta-type in CORBA called *Component* which supports multiple independent interfaces (current CORBA only supports multiple interfaces by inheritance). A *CORBAcomponents container* provides infrastructure to navigate among these interfaces.

The CORBAcomponents container environment is persistent, transactional, and secure. For the developers, these functions are pre-packaged, and provided at a higher level of abstraction than the CORBAservices provide. This helps put CORBA middleware development within the domain of business application development, thus allowing application developers with little knowledge of CORBA to use it in a way they can understand.

Enterprise Java Beans (EJBs) will act as CORBAcomponents integrated in the container. They will have CORBAcomponent-style remote interfaces, defined by CORBA IDL. The specification states that CORBAcomponent implementation may be packaged and deployed using XML-based tools, and also defines a multi-platform software distribution format.

Specified in [33], *Component Scripting* is a standard scripting language used to wire all components together, through which users can easily modify or upgrade applications constructed utilizing the language. The specification maps component assembly to a number of widely used scripting languages.

3.5 MICO [44]

The distributed DISIMA implementation uses a public domain implementation of CORBA 2.3.1, called MICO. The acronym MICO expands to "*M*ICO *I*s *CO*RBA". It is a freely available and fully compliant implementation of the CORBA standard. MICO has become fairly popular, and has been designated as CORBA-compliant by the OMG. MICO is implemented in C++.

The current version of MICO includes the following features as indicated in [44]:

- IDL to C++ mapping
- Dynamic Invocation Interface (DII)
- Dynamic Skeleton Interface (DSI)
- graphical Interface Repository browser that allows invocation of arbitrary methods on arbitrary interfaces
- Interface Repository (IR)
- IIOP as native protocol (ORB prepared for multiprotocol support)
- Support for nested method invocations
- *Any* type offers an interface for inserting and extracting constructed types that were not known at compile time
- Full BOA implementation, including all activation modes, support for object migration, and the implementation repository
- BOA can load object implementations into clients at runtime using loadable modules
- Portable Object Adapter (POA)
- Objects by Value (OBV)
- Support for using MICO from within X11 applications (Xt, Qt, and Gtk) and Tcl/Tk

- Dynamic Any
- Interceptors
- Support for secure communication and authentication using SSL
- CORBA Object Services:
  - Interoperable Naming service
  - Trading service
  - Event service
  - Relationship service
  - Property service
  - Time service

MICO works on the following platforms:

- Solaris 2.5, 2.6, and 7 on Sun SPARC (egcs 1.x, native C++ compiler)
- AIX 4.2 on IBM RS/6000 (egcs 1.x, native C++ compiler)
- Linux 2.x on Intel x86 (egcs 1.x)
- Digital Unix 4.x on DEC Alpha (egcs 1.x)
- HP-UX 10.20 on PA-RISC (egcs 1.x, native C++ compiler)
- Ultrix 4.3 on DEC Mips (egcs 1.x)
- Linux 2.x on DEC Alpha (egcs 1.x)
- SGI-IRIX on DEC Mips (egcs 1.x, native C++ compiler)
- PowerMax OS (native C++ compiler)
- Windows 95/NT (Cygnus CDK beta19)
- Windows 95/NT (Visual-C++ 5+SP3 and 6+SP1)
- FreeBSD 3.x on Intel x86 (egcs 1.x)

MICO is free software. Though MICO does not provide much support in the development of database systems, such as Query Service and Transaction Service, it does have as many features as a fully CORBA compliant implementation, which is one reason for its selection in the distributed DISIMA implementation.

Chapter 4

CGI

Besides the OMA, another infrastructure, that is used extensively in the distributed DISIMA implementation, is the Common Gateway Interface (CGI [5, 9]). This is useful in distributed DISIMA because the CGI provides the communications, via the Web server, between Web-based DISIMA clients (VisualMOQL), and CORBA-based DISIMA servers in the distributed DISIMA architecture (discussed in detail in Chapter 5).

4.1 Background

A major benefit of the *World Wide Web (Web or WWW)* is that Web authors can provide users with interlinked hypertext documents on a diverse range of topics. Users can then access a variety of information from anywhere in the world, through Web browsers such as Netscape Navigator or Microsoft Internet Explorer. Web documents are normally delivered by *HTTP* (*HyperText Transfer Protocol* [10]) servers running *httpd* (*HTTP daemon*). Servers and browsers communicate through the Internet, which connects a large network of computers worldwide.

Web documents are usually written in *HyperText Markup Language (HTML* [11]) and stored statically, as text files, on the server's disk. Such simple static hypertext documents can carry lots of information to users, but their limitations prevent them from dealing with increasing numbers of "interactive", dynamically generated, Web documents. For example, what if an author wishes to provide flight schedule information which changes over time? Or, as in DISIMA, one wishes to provide an interactive interface that allows users to explore the database by specifying the query?

There are mainly two approaches to generating *dynamic documents*. One is the *server-side include* mechanism, which aims to assemble a single large HTML document from several smaller documents. Unfortunately, the performance problems [5] of this mechanism make it unpopular. The most commonly used approach is *Common Gateway Interface (CGI)*, a standard for the construction of completely dynamic documents by external programs running on the server system, in a platform independent manner.

4.2 The CGI Standard

The CGI standard was developed jointly by the US *National Center for Supercomputing Applications (NCSA)* and the *European Laboratory for Particle Physics (CERN)* in 1993, and has since been widely utilized in the World Wide Web. It is a simple standard in which external applications interact with information servers (such as HTTP servers), and regulate the environment where the external applications execute. A plain HTML document retrieved by the Web server is static (i.e., existing as a text file in a constant state), while a CGI program is executed by the server as per users' requests in real-time, so that it can output dynamic documents.

The goals of CGI can be summarized as follows:

- To provide a consistent, standard interface between the Web server and the external application.
- To make sure that user input will not be lost due to the limitations of the server operating system.
- To provide the external program with as much information as possible about the server, the browser, and the user.
- To keep the CGI standard as simple as possible to simplify the development of the CGI application.

Figure 4.1 describes how the browser, the Web server, and the CGI program collaborate to provide real-time dynamic information to the user.

Let's take our DISIMA project as an example. We have a Web based VisualMOQL query interface, and an image DBMS running on the UNIX system. How

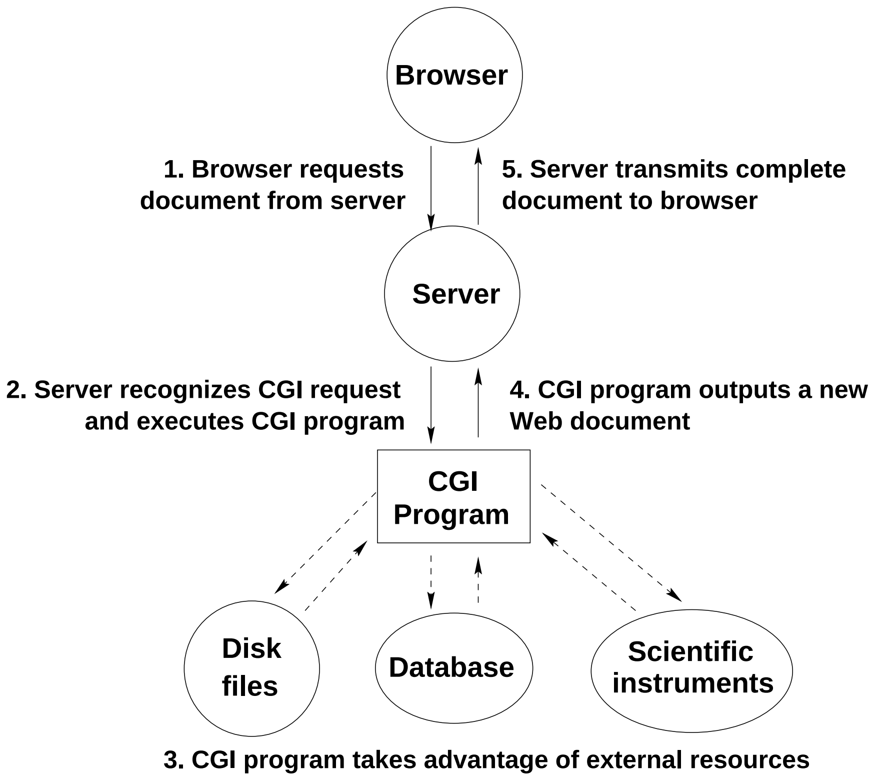

Browser
1. Browser requests document from server
5. Server transmits complete document to browser
Server
2. Server recognizes CGI request and executes CGI program
4. CGI program outputs a new Web document
CGI Program
Disk files
Database
Scientific instruments
3. CGI program takes advantage of external resources

Figure 4.1: The CGI Mechanism (Taken from [5])

can we connect these two components in order to make the image DBMS accessible to users from all over the world? In addition to a Web server, we need to create a CGI program that the Web server will execute to: transmit the query to the database engine, receive the query results back, and display them to the user through the browser.

Since a CGI program is executable, for security concerns, it needs to be located in a special directory so that the Web server knows where to execute the CGI program. The webmaster usually takes charge of this directory, prohibiting unauthorized users from creating CGI programs.

A CGI program can be written in virtually any language that allows it to be executed on the system, such as: C/C++, Fortran, PERL, TCL, any Unix shell, Visual Basic, and AppleScript. The CGI application developed in programming languages, such as C/C++ and Fortran, needs to be compiled before it runs; while the application in scripting languages, such as PERL, TCL, or a Unix shell, can run directly. We chose C/C++ as the CGI programming language for DISIMA because of its performance advantages [5] over others.

The browser sends its requests to the Web server by using HTTP, and the server executes CGI programs based on the requests from the browser. There are several kinds of HTTP requests (so called *methods*), two of which, `GET` and `POST`, concern CGI programs:

- The `GET` method – The browser requests a document usually without submitting any other input.
- The `POST` method – This method is used to deliver information from the browser to the server. Most likely, the information sent is a form submission.

The standard output of CGI programs can be either a new valid Web document, an error code, or a redirection to another document. CGI programs can obtain input information from environment variables shared with the Web server, data passed on the command line of the browser, or the standard I/O calls.

4.3 Limitations

The simplicity of the CGI standard comes at a price. One major drawback is that every time a CGI program is requested, the operating system must create a new, distinct process for the CGI program and set the various CGI environment variables appropriately. This leads to the following issues:

- Each process consumes resources within the server's operating system. The fixed cost associated with every execution of a CGI program is a waste of resources.

- If a CGI program process takes a long time to execute, the user will have to wait in front of the browser until the CGI program returns something, or the user must explicitly quit the request.

- Since there are no direct links between two CGI processes, it is difficult to keep track of the state of communications between the user and the server.

Chapter 5

Distributed DISIMA

As we mentioned in Chapter 1, the DISIMA project addresses the development of a distributed, interoperable image database management system with the following features:

1. object oriented approach to image data management
2. use of image processing and indexing techniques for efficient querying and access to image databases
3. access to distributed (and possibly heterogeneous) image storage systems

This project addresses the third point. Two steps are followed in reaching this goal:

- First we put the single DISIMA into the OMA; specifically, we create a CORBA-based single DISIMA system.
- Then, based on the prototype, we put multiple databases into the structure and deal with interoperability issues that arise.

All the distribution work will be implemented using C/C++ language, as the DISIMA prototype is implemented in C/C++ and MICO only supports C/C++. Before going any further on the design, we discuss how the OMA can fit into our requirements, and some design issues that need to be considered.

5.1 CORBA and Database Interoperability

As an object-oriented distributed computing platform, OMA, and in particular CORBA, can be helpful for database interoperability in terms of managing heterogeneity. In a multidatabase system implementation, the major issue is how to deal efficiently with heterogeneity which, basically, exists at four levels: the hardware and operating system level (or jointly called the platform level), the communication level, the DBMS level, and the semantic level. CORBA deals primarily with platform and communication heterogeneity. It provides implementation and location transparency, which enables a client to access an object through the object's interfaces defined by IDL and its object reference, and independent of the platform and the location where the object resides, or the communication protocol between the client and the object. The DBMS-level heterogeneity is among DBMSs based on different data models and query languages. Semantic heterogeneity addresses incorporation of databases with different schemas, and includes schema conflicts and data conflicts. One possible solution to handle the DBMS level and semantic level heterogeneity is to develop a global layer that includes the global level DBMS functionality. One issue with which CORBA cannot be directly helpful is semantic heterogeneity.

Using CORBA as the infrastructure affects the upper layers of a multidatabase system, since CORBA and CORBAservices together provide basic database functionality for managing distributed objects. The most important database-related services included in CORBAservices are: Transaction Service, Backup and Recovery Service, Concurrency Service, and Query Service. If they are available in the ORB implementation used, it is possible to develop the global layers of a multidatabase system on CORBA, mainly by implementing the standard interfaces of these services for the objects involved. For example, when using a Transaction Service, implementation of a global transaction manager occurs by implementing the interfaces defined in the Transaction Service specification for the involved DBMSs. Unfortunately, most commercial ORBs do not support these services; nor does MICO ORB.

In this section, we discuss some of the design issues that must be resolved in order to use CORBA for database interoperability. This discussion is based on [1].

Object Granularity. A fundamental design issue is the granularity of the CORBA objects. In registering a DBMS to CORBA, a row in a relational DBMS, an object or a group of objects in an object DBMS, or a whole DBMS, can be an individual CORBA object. The advantage of fine granularity objects is the finer control they permit. However, in this case, all the DBMS functionalities (e.g., querying and transactional control) needed to process and manage these objects have to be supported by the global system level (i.e., the multidatabase system). If, on the other hand, a whole DBMS is registered as a CORBA object, the functionality needed to process the entities is left to that DBMS.

Another consideration with regard to granularity has to do with the capabilities of the particular ORB being used. In the case of ORBs that provide BOA, each insertion and deletion of classes necessitates recompiling of the IDL code and rebuilding the server. Thus, if the object granularity is fine, these ORBs incur significant overhead. A possible solution to this problem is to use DII. This prevents recompilation of the code and rebuilding of the server, but suffers the run-time performance overhead discussed earlier.

Object Interfaces. A second design issue is the definition of interfaces to the CORBA objects. Most commercial DBMSs support the basic transaction and query primitives, either through their Call Level Interface (CLI) library routines or their XA interface[1] library routines. This property makes it possible to define a generic database object interface through CORBA IDL, to represent all the underlying DBMSs. CORBA allows multiple implementations of an interface. Hence, it is possible to encapsulate each of the local DBMSs by providing a different implementation of the generic database object.

Association Mode. The association mode between a client request and server method is a third design issue. As specified earlier, CORBA provides three alternatives for handling this: one interface to one implementation; one interface to one of many implementations; and one interface to multiple implementations. The choice of alternative is dependent both on the data location and the nature of the database access requests. If the requested data is contained in one database, then it is usually sufficient to use the second alternative, and choose the DBMS that manages that data — since DBMSs registered to CORBA provide basic transaction management and query primitives for all the operations the interface definition specifies. If the request involves data from multiple databases, then the third alternative needs to be chosen.

[1]The XA interface is defined in the X/Open Distributed Transaction Processing model proposed by the Open Group, a vendor consortium. The model comprises four components: Application Programs, Resource Managers, Transaction Managers, and Communication Resource Manager. The *XA interface* is a specification that describes the protocol for transaction coordination, commitment, and recovery between Resource Managers and Transaction Managers. [2]

Call Mode. As discussed earlier, CORBA 3 defines four basic call communication modes between a client and a server: synchronous, deferred synchronous, one-way, and asynchronous. For objects of an interoperable DBMS, synchronous call mode is generally sufficient. Deferred synchronous mode or the asynchronous (peer-to-peer) approach should be used when parallel execution is necessary. For example, in order to provide parallelism in query execution, the global query manager of a multidatabase system should not wait for the query to complete after submitting it to a component DBMS.

Concurrently Active Objects. Some of the objects in a multidatabase system need to be concurrently active. This can be achieved either by using threads on a server that uses a shared activation policy, or by using separate servers activated in the unshared mode for each object. Otherwise, since a server can only give service for one object at a time, client requests to other client requests to the objects owned by the same server should wait for the current request to complete. Further, if the server keeps transient data for the object throughout its life cycle, all requests to an object must be serviced by the same server. For example, if a global transaction manager is activated in shared mode, it would be necessary to preserve the transaction contexts in different threads. However, if the global transaction manager is activated in unshared mode, the same functionality can be obtained with a simpler implementation, at the cost of having one process for each active transaction.

5.2 The Single Site DISIMA

Let us first take a look at current single site DISIMA implementation structure. Figure 5.1 shows the original version of single site DISIMA implementation structure. Anything inside the dotted line circle is physically located in the same machine, called “Darwell”. After the user composes a query using VisualMOQL, and submits the query, the Web browser passes the query to the Web server. Upon receiving the request, the Web server executes a CGI program (the DISIMA program), which opens the image database, parses the query, executes the query against the database, sends the result image(s) back to the Web server through the CGI, and closes the database. The Web server delivers generated documents to the browser.

The implementation structure is simple and clear. However, the underlying problem is that since the DISIMA program is itself a CGI program, every time the user sends requests, the Web server needs to create a separate DISIMA instance for

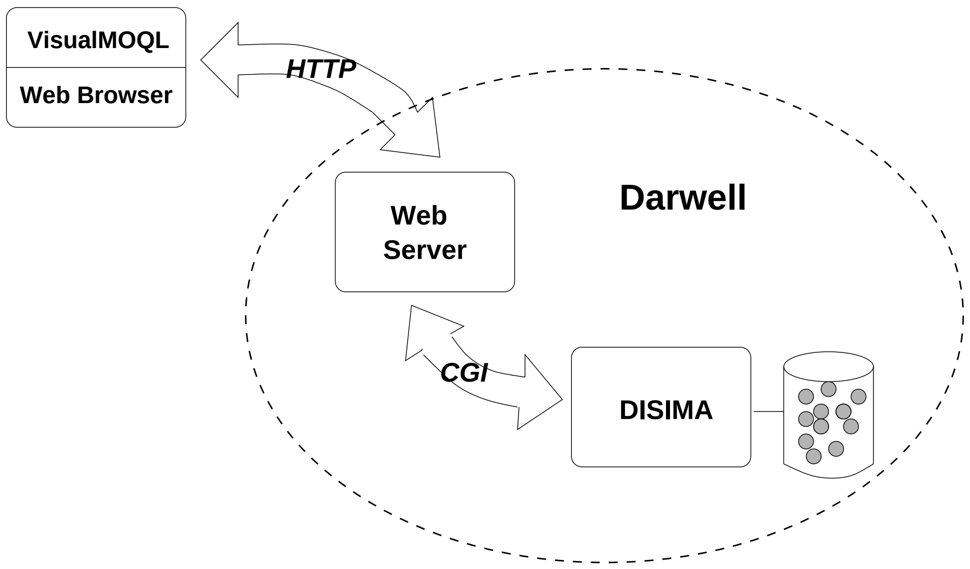


Figure 5.1: The Original Single DISIMA Implementation Structure

each request. For each instance, the image database needs to be initialized, opened, and closed repeatedly. As we discussed before, it is a waste of time and resources of the operating system to use complicated CGI programs, such as DISIMA, in this case. A better implementation structure is to make the DISIMA program run as a service (daemon). Thus, the image database only needs to be initialized and opened once. Upon the user's requests, the Web server launches a small CGI program, which passes the query to the DISIMA daemon. The DISIMA daemon executes the query against the image database, and returns the result images back to the user through the CGI program.

Adapting ObjectStore ObjectForms [25], the latest single site DISIMA implementation structure, is illustrated in Figure 5.2. ObjectStore *ObjectForms* provides a communication channel between the Web server and the DISIMA — an ObjectStore application. Instead of calling the DISIMA directly, the Web server executes an ObjectForms CGI program, which sends the query to the DISIMA daemon and waits for a response. The DISIMA program is modified to use ObjectForms API to create and initialize a service which allows the DISIMA to respond to requests from the Web server through the ObjectForms communication channel. The communication channel is transparent to users and developers.

ObjectForms solves the problems we previously encountered. Only one in-

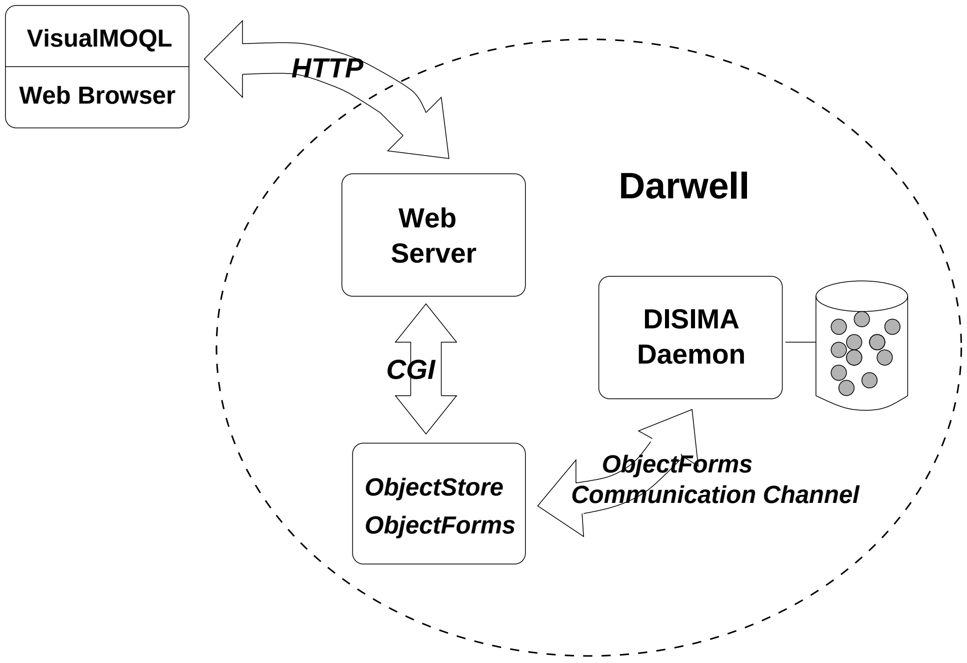


Figure 5.2: The Single DISIMA Implementation Structure

stance of the DISIMA program exists in the operating system. As long as the image database is up, the DISIMA program only initializes and opens the database once, and deals with the queries passed along from the ObjectForms. This solution is perfect for the single site DISIMA structure.

5.3 The CORBA-based Single Site DISIMA

As we discussed before, when it comes to CORBA and database interoperability, the primary issue is granularity. We decided to register an entire DISIMA as a CORBA object for the following reasons:

- MICO ORB does not provide database-related object services, such as Transaction Service, Backup and Recovery Service, Concurrency Service, and Query Service. Only two object adapters are included: BOA and POA. In fact, database-related object services, and the object-oriented DBMS (OODBMS) object adapter, are not implemented in most available commercial ORBs.
- Since the DISIMA itself is a fully functional DBMS, we can leave all database related functionalities to the DISIMA.

ObjectStore ObjectForms is not suitable for the CORBA-based implementation because the communication channel it provides is transparent to the developers, and it requires that all the applications linked to it are ObjectStore applications — which does not necessarily happen in the heterogeneous environment. Instead, we create a small CGI program, *Query Agent*, which is also a CORBA client, to provide the same functionalities.

If MICO ORB had Query Service, we could register each image in the image database as a CORBA object. Thus, with Query Services, the Query Agent could conduct finer query operations on the result images (a collection of CORBA objects) coming from the DISIMA. Furthermore, it could directly manipulate images in the image database using some query languages (such as MOQL) without going through the DISIMA.

The CORBA-based single DISIMA architecture is depicted in Figure 5.3. Objects in ellipse shape are CORBA objects. Since all the CORBA objects are known, and the interface of the database object can be defined on compile time, stub-style (static) interface invocation is sufficient. The input data is a query string and the

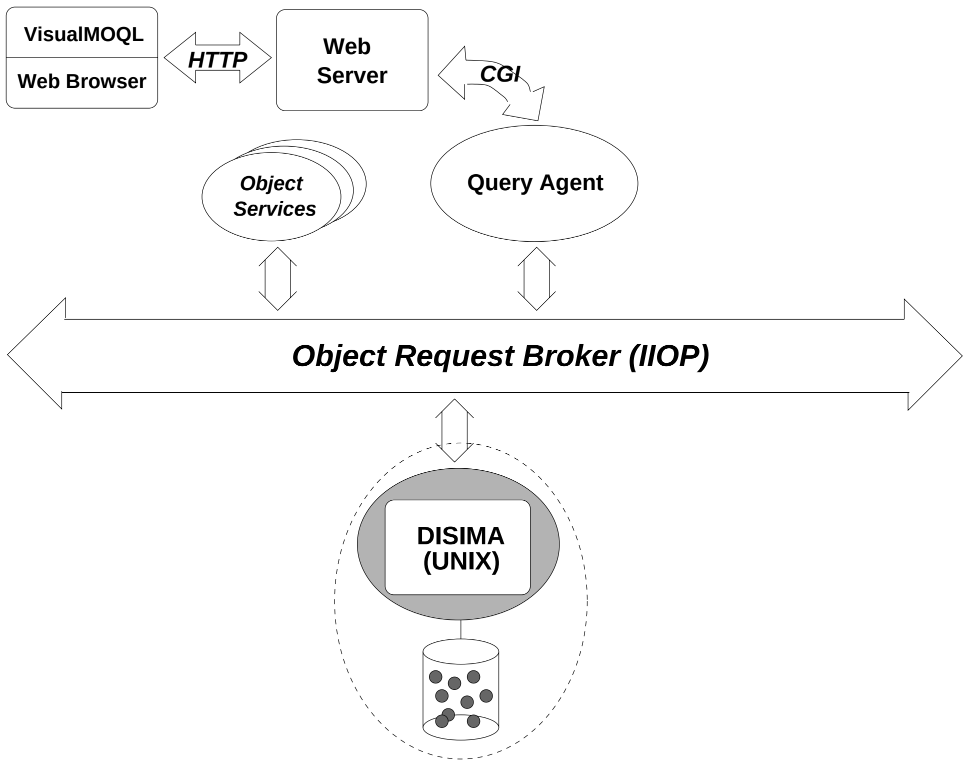


Figure 5.3: The CORBA-based Single DISIMA Architecture

output data is a set of thumbnails with related information, or an enlarged image. The following is the major part of the IDL file.

```
module DISIMA {

 exception SyntaxError {
   unsigned short position;
   string errMessage;
 };

 typedef sequence<octet> thumbnailStreamType;

 typedef sequence<octet> imageStreamType;

 struct thumbnailType {
     string serverId;
     string thumbnailLabel;
     string thumbnailSuffix;
     string thumbnailId;
     thumbnailStreamType thumbnailStream;
     float similarity;
 };

 struct imageType {
     string serverId;
     string imageLabel;
     string imageSuffix;
     string imageId;
     imageStreamType imageStream;
 };

 typedef sequence<thumbnailType> Thumbnails;

 typedef imageType AImage;

 interface QueryAgent {
     Thumbnails getThumbnails ( in string queryString )
                       raises ( SyntaxError );
     AImage getImage ( in string queryString )
37                        raises ( SyntaxError );
38  };
39 };
```

Lines 3-6 define the content of any exception that may be raised. Lines 8 and 10 define the types for thumbnail stream and image stream, which are unbounded octet arrays. Lines 12-19 define the structure of the thumbnail type, which includes the server Id indicating which database this thumbnail comes from; the thumbnail label (name); the thumbnail suffix (image type); the thumbnail Id (an octet sequence that uniquely identifies the thumbnail in the database it comes from; in our case, it is an object Id in ObjectStore); the thumbnail stream (real thumbnail data); and the similarity between the target thumbnail and the query image. Lines 21-27 define the content of the image type. Lines 29 and 31 define the return type of the query, i.e., a set of thumbnails or an image.

Lines 33-38 define an interface called `QueryAgent` which includes two operations (methods) on a CORBA database object:

- `getThumbnails` – The input parameter is a MOQL query string that will be passed along to the database object; the output is a set of result thumbnails matching the query.
- `getImage` – The input parameter includes all information to get an image from a particular database; the output is an image.

Both these operations may raise exceptions, defined in lines 3-6.

The IDL file is compiled by the IDL compiler and mapped into C++ language files, including client stub (integrated into the Query Agent and used by it to invoke methods on remote database objects), server skeleton (class declarations for the DISIMA object implementation), and other supporting codes.

The simplified process flow is shown in Figure 5.4. A Query Agent instance is created by the Web server.

The main functionalities of the Query Agent are as follows:

- Get the query string from the environment variable `QUERY_STRING` (containing any information submitted by the browser as a result of a `GET` method) using $getenv()$ function, and validate the query string. A typical query string can be in either of the following forms:

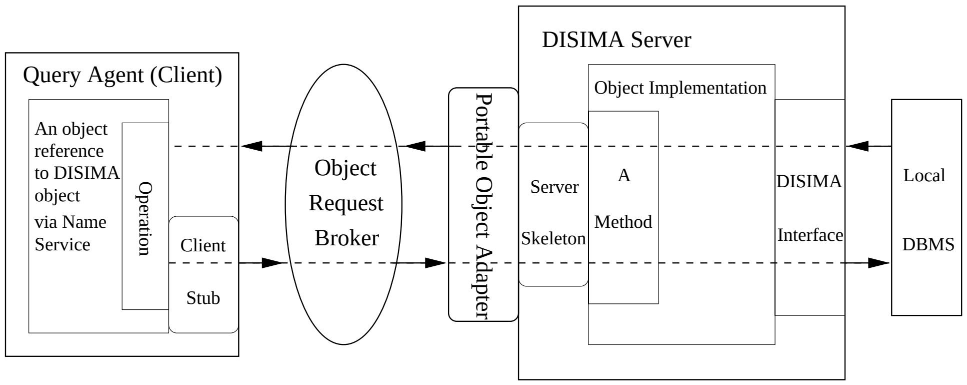


Figure 5.4: Invoke an operation to DISIMA object via ORB (Adapted from [1])

```
String=SELECT|m|FROM|Image|m;&num=25&threshold=0.95
                        or
Image=%3C%2Fdarwell%2Fvar%2Ftmp%2Fdemo_c%2Edb+%7C+2+%7C+5e808%3E&
      size=13258&server=DB_darwell
```

- Initialize ORB and connect to naming service

```
// ORB initialization
CORBA::ORB_var orb =
    CORBA::ORB_init( argc, argv, "mico-local-orb" );
...

// Get reference to initial naming context
CORBA::Object_var nobj =
    orb->resolve_initial_references ("NameService");
...

// Narrow
CosNaming::NamingContext_var nc =
    CosNaming::NamingContext::_narrow (nobj);
...
```

- Using Naming Service, get the object reference to the remote DISIMA server object

```
// Construct a server object name context
```

```
CosNaming::Name name;
name.length (1);
name[0].id = CORBA::string_dup (serverName);
name[0].kind = CORBA::string_dup ("");
...

// Resolve the name and get the object reference to the server object
CORBA::Object_var obj = nc->resolve (name);
...

// Narrow
QueryAgent_var client = QueryAgent::_narrow( obj );
...
```

- Depending on the content of the query string, call `getThumbnails` method in synchronous mode to get a set of thumbnails, or call `getImage` method in synchronous mode to get the image data.
- Construct a dynamic HTML file and send it back to the Web server as a standard output

The Naming service is the simplest and most basic of the standardized CORBA common object services, and is itself an object. Through a running naming service daemon, which is registered with the CORBA implementation repository, the DISIMA server object can bind its names to its object reference, and the Query Agent can query the name server to resolve the object reference.

The IDL-wrapped DISIMA server is registered with the CORBA implementation repository, using shared persistent activation policy, which means the server is started manually. The MICO ORB automatically delivers the request from the Query Agent to the DISIMA server, specifically on a particular method. The implementation codes for each method declared in the IDL file are included in a class called `QueryAgent_impl`, as follows:

```
class QueryAgent_impl : virtual public POA_QueryAgent {
public:
  Thumbnails *getThumbnails( const char * queryString ) {...}
  AImage *getImage( const char * queryString ) {...}
  ...
}
```

Inside each method, the query string is sent to the local DBMS through its CLI, and results are composed and returned to the invoker. Based on the user's request, the appropriate method will be invoked.

The CORBA DISIMA server includes the following functions:

- Initialize the image database through the interfaces.
- Initialize ORB and root POA, and create the servant that provides the services to the Query Agent.

```
// ORB initialization
CORBA::ORB_var orb;
...

// Obtain a reference to the RootPOA and its Manager
CORBA::Object_var poaobj =
    orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var poa =
    PortableServer::POA::_narrow (poaobj);
PortableServer::POAManager_var mgr = poa->the_POAManager();
...

// Create and activate the servant
QueryAgent_impl *servant = new QueryAgent_impl;
QueryAgent_var oid = servant->_this();
...
```

- Connect to naming service and bind its name to its object reference

```
// Connect naming service
CosNaming::NamingContext_var nc;
...

// Bind
appName.length (1);
appName[0].id = CORBA::string_dup (serverName);
appName[0].kind = CORBA::string_dup ("");
```

```
...

nc->rebind (appName, oid);
...
```

- After the initialization, the server waits for requests, or for the termination signals which are handled by the function `sigHandler`. One of the important tasks is to unbind the server's name so that the Query Agent can know that the DISIMA server is not available anymore.

  ```
  nc->unbind(appName);
  ```

The CORBA-based, single-site DISIMA prototype is a milestone for this project because it is the first time that a CORBA-based DBMS is actually implemented.

5.4 The Distributed DISIMA

In our distributed DBMS environment, the query results may come from multiple databases, and these databases are potentially heterogeneous. Therefore, the best choice of association mode between a client request and a server method is one interface to multiple implementations.

We can still use the same IDL interface `QueryAgent`, as our generic database object interfaces to all local DBMSs. Each local DBMS provides a different implementation of that generic interface.

The distributed DISIMA architecture is described in Figure 5.5. We use the query language (MOQL), the data model and schema of the DISIMA system as our global query language, data model and schema, respectively. Note that we add a *Local Query Agent* for each CORBA DBMS server object to deal with the database system (e.g., data models, query language) and semantic (schema) heterogeneity — i.e., the third and fourth level heterogeneity from Section 5.1. The implementation of the Local Query Agent is beyond the scope of this project. The functions of the Local Query Agent would at least include:

- Mapping the global query language (MOQL) into local query languages;
- Translation of the global data model into local data models; and
- Resolving schema conflicts between global and local schemas.

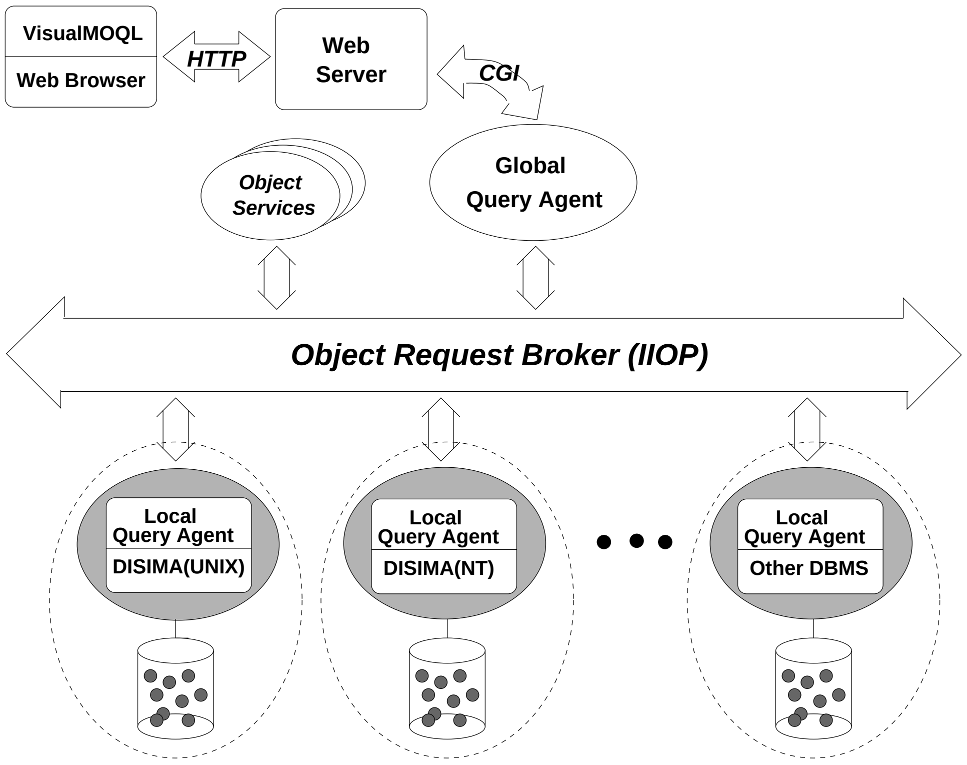


Figure 5.5: The Distributed DISIMA Architecture

The *Global Query Agent* is derived from the *Query Agent* in Figure 5.3. Besides all the functions of the Query Agent, the Global Query Agent sends the query request to all database server objects, and integrates result thumbnails from available database servers. For example, the user sends out the query with the maximum number of returned images, N. The Global Query Agent sends the same query, with the same number N, to all the servers. If the total number of images (in the form of thumbnails) returned from available servers is larger than N, the Global Query Agent sorts the thumbnails according to their similarities, in descending order, and returns the first N thumbnails to the user; otherwise, the Global Query Agent sorts the thumbnails and returns all thumbnails back to the user. The Global Query Agent would also be responsible for resolving semantic heterogeneity issues among results returned by individual servers. Since this is beyond the scope of this project, the issue is not discussed further.

At this point, one important issue needs to be addressed: whether the synchronous communication mode is still suitable for the distributed environment. In the synchronous communication mode, the client waits until it gets a response from the server. If the server shuts down normally, it will unbind its name on naming service daemon and update the implementation repository; thus, the client will be informed, and won't call this server. However, if the server shuts down abnormally, this will make the name binding, and the information in the implementation repository, out of date. The client that invokes the server will keep waiting forever, which is not acceptable. The same problem happens when the network is congested (which occurs frequently on the Internet), and the server cannot send its response back promptly. Clearly the synchronous communication mode no longer satisfies our requirements.

We cannot use the asynchronous communication mode because the current version of MICO ORB does not support it. We could use one-way callbacks, but one-way operations have "best effort" semantics, which means that one-way calls are not guaranteed to be delivered. Issues like object reference equality, callback persistence, callback failure, and scalability bring new complicated requirements to our design and implementation. Detailed discussion of one-way callbacks can be found in [21].

The communication problem is solved in the current implementation by means of the Event Service. The Event Service allows an application to use a decoupled communications model (specifically, deferred synchronous mode), rather than strict client-to-server synchronous request invocations. In the Event Service model, *suppliers* produce events, and *consumers* receive them, through an *event channel*.

Event channels allow multiple suppliers and consumers to be connected to them. It is not necessary for suppliers to know about consumers, or vice versa.

The Event Service provides the *push* model and the *pull* model for event delivery, illustrated in Figures 5.6 and 5.7, respectively. Using the push model, suppliers (clients) push events to the event channel, and the event channel pushes events to consumers (servers). With the pull model, consumers (clients) pull events from the event channel, and the event channel pulls events from the suppliers (servers). Besides, event channels support a *hybrid push/pull* model (suppliers push events to the channel and consumers pull events from the channel) and a *hybrid pull/push* mode (the channel pulls events from suppliers and pushes events to consumers). The supplier and the consumer never contact the event channel directly. Instead they interact with *proxy* interfaces, which represent the actual supplier and the actual consumer.

The type of data that is carried on through the event channel is called *Any*, which is an IDL type that provides a universal type that can hold a value of arbitrary IDL type. Thus, we can send and receive values whose types are not fixed at compile time, through the event channel.

For our distributed DISIMA architecture, we need two event channels: the *Request Event Channel* and the *Result Event Channel*. We use the push model for the request event channel, and the hybrid push/pull model for the result event channel (Figure 5.8). The global query agent still uses Naming Service to fetch the image from the particular server.

For the MOQL query string, the global query agent pushes the query with a certain tag to the requests event channel (we use the dotted-decimal IP address of the client browser to identify itself from other query agent instances). Based on the above changes, we modify the IDL file as follows:

```
module DISIMA {

 exception SyntaxError {
   unsigned short position;
   string errMessage;
 };

 typedef sequence<octet> thumbnailStreamType;

```

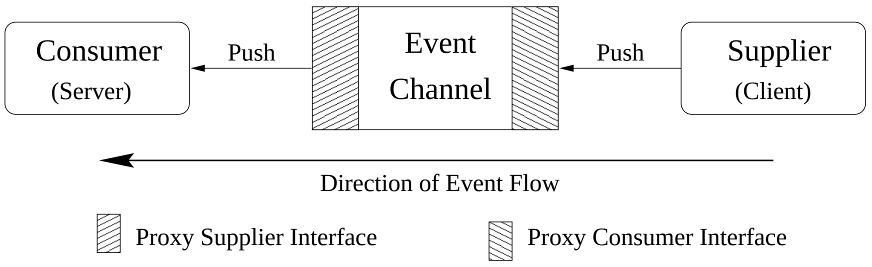


Figure 5.6: Push-style Event Delivery Model (Adapted from [21])

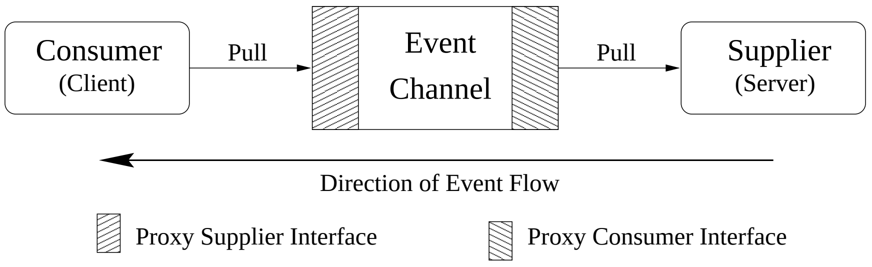


Figure 5.7: Pull-style Event Delivery Model (Adapted from [21])

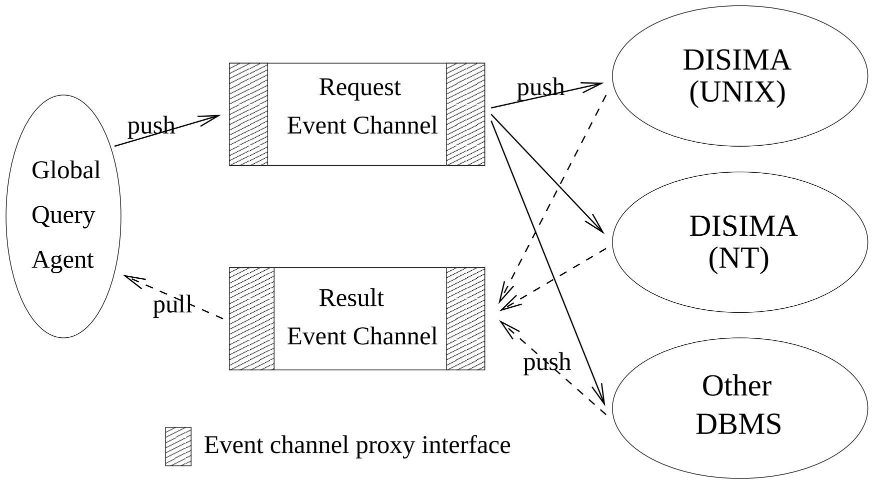


Figure 5.8: Introduce Event Channels into the DISIMA Architecture

```
typedef sequence<octet> imageStreamType;

struct thumbnailType {
    string serverId;
    string thumbnailLabel;
    string thumbnailSuffix;
    string thumbnailId;
    thumbnailStreamType thumbnailStream;
    float similarity;
};

struct imageType {
    string serverId;
    string imageLabel;
    string imageSuffix;
    string imageId;
    imageStreamType imageStream;
};

typedef sequence<thumbnailType> thumbnailsType;

struct Thumbnails {
    string remoteAddr;
    thumbnailsType thumbnails;
};

struct Requests {
    string remoteAddr;
    string queryString;
};

typedef imageType AImage;

interface QueryAgent {
    Thumbnails getThumbnails ( in Requests requests )
                      raises ( SyntaxError );
    AImage getImage ( in string queryString )
                      raises ( SyntaxError );
};
};
```

The major difference between the new IDL file, and the original IDL file in Section 5.3, is that we attach a string type tag `remoteAddr`, which is an IP address, to the input/output parameter of the method `getThumbnails`. Lines 29-34 define the return type `Thumbnails` of the method, i.e., a set of thumbnails with a tag. Lines 36-39 define the input type of the method, i.e., a query string with a tag. The global query agent can get the IP address of the client browser from the environment variable `REMOTE_ADDR` using *getenv()* function.

The global query agent does not have to worry about how many DBMS servers are available, and what they are. On the other side of the channel, the requests are pushes to all the servers that are linked to the request event channel. Each server pushes back its results through the result event channel, while the global query agent keeps trying to pull the results from the result event channel, after it sends the query request. When the global query agent detects any result from the event channel, it uses the tag attached to the results to identify whether the results are desired. If so, it sends the results back to the client; otherwise, it ignores that result.

The global query agent uses the following major codes to connect to the request event channel and push the request.

```
// Get the object reference to request event channel
CosNaming::Name name;
name.length (1);
name[0].id = CORBA::string_dup ("RequestsEventChannel");
name[0].kind = CORBA::string_dup ("");
CORBA::Object_var requests_obj = nc->resolve (name);
...

// Connect to request event channel
CosEventChannelAdmin::EventChannel_var requests_event_channel;
CosEventChannelAdmin::SupplierAdmin_var supplier_admin;
CosEventChannelAdmin::ProxyPushConsumer_var proxy_consumer;
requests_event_channel =
  CosEventChannelAdmin::EventChannel::_narrow (requests_obj);
supplier_admin = requests_event_channel->for_suppliers ();
proxy_consumer = supplier_admin->obtain_push_consumer ();
...

// Construct a request
```

```
  Requests *requests = new Requests;
  requests->queryString = queryString;
  requests->remoteAddr = remoteAddr;

  // Insert the query data into a CORBA::Any
  CORBA::Any anyRequests;
  anyRequests <<= (Requests *) requests ;

  // Push the event to the event channel;
  proxy_consumer->push (anyRequests);
  ...
```

The database server includes a class, `Consumer_impl`, in which the push method retrieves the query from *any* data through the request event channel, sends the query to the local DBMS through its CLI, gets the results, and pushes the results back to the global query agent through the result event channel.

```
class Consumer_impl : virtual public POA_CosEventComm::PushConsumer{
public:
  ...

  void push (const CORBA::Any& data) {
    ...

    // (*retThumbnails) are query results
    Thumbnails *retThumbnails;
    ...

    // Insert the query results into a CORBA::Any
    CORBA::Any anyOut;
    anyOut <<= (Thumbnails *)retThumbnails;

    // Push the event to Results Event Channel
    proxy_consumer->push (anyOut);
    ...

  }
  ...
};
```

The try-pull procedure in the global query agent is actually a loop procedure, which is depicted in Figure 5.9. We can specify how long the global query agent needs to wait for a database server to response.

Figure 5.10 shows an example of distributed query results. It shows that currently there are four active database servers and nine images (in the form of thumbnails) that match the query coming from two servers (DB_delia and DB_sakwatamau). The server name of each image is indicated below the image label.

5.5 Implementation Issues

To summarize the design and implementation of the distributed DISIMA system, we need to point out three issues for further improvement.

CGI Because we use CGI programming, the distributed system is subject to the shortcomings of CGI, e.g., for each VisualMOQL user, the Web server invokes a separate global query agent instance.

Image Integration The global query agent simply integrates all the images (in the form of thumbnails) coming from all database servers, without considering whether two images from different servers may have the same contents.

IP Address The whole distributed system has one result event channel. Each global query agent instance needs to know whether the event it pulls from the channel is what it wants. Assume that, with only one user for each machine, we use the IP address to identify each event. However, if a proxy server is used by a particular set of users, many machines may appear to have one IP address. Two users in such a group may get the same results if they submit two different queries at the same time. If two users on the same client machine issue two different queries at the same time, or if one user issues two queries (by instantiating multiple VisualMOQL interfaces), the same problem would arise.

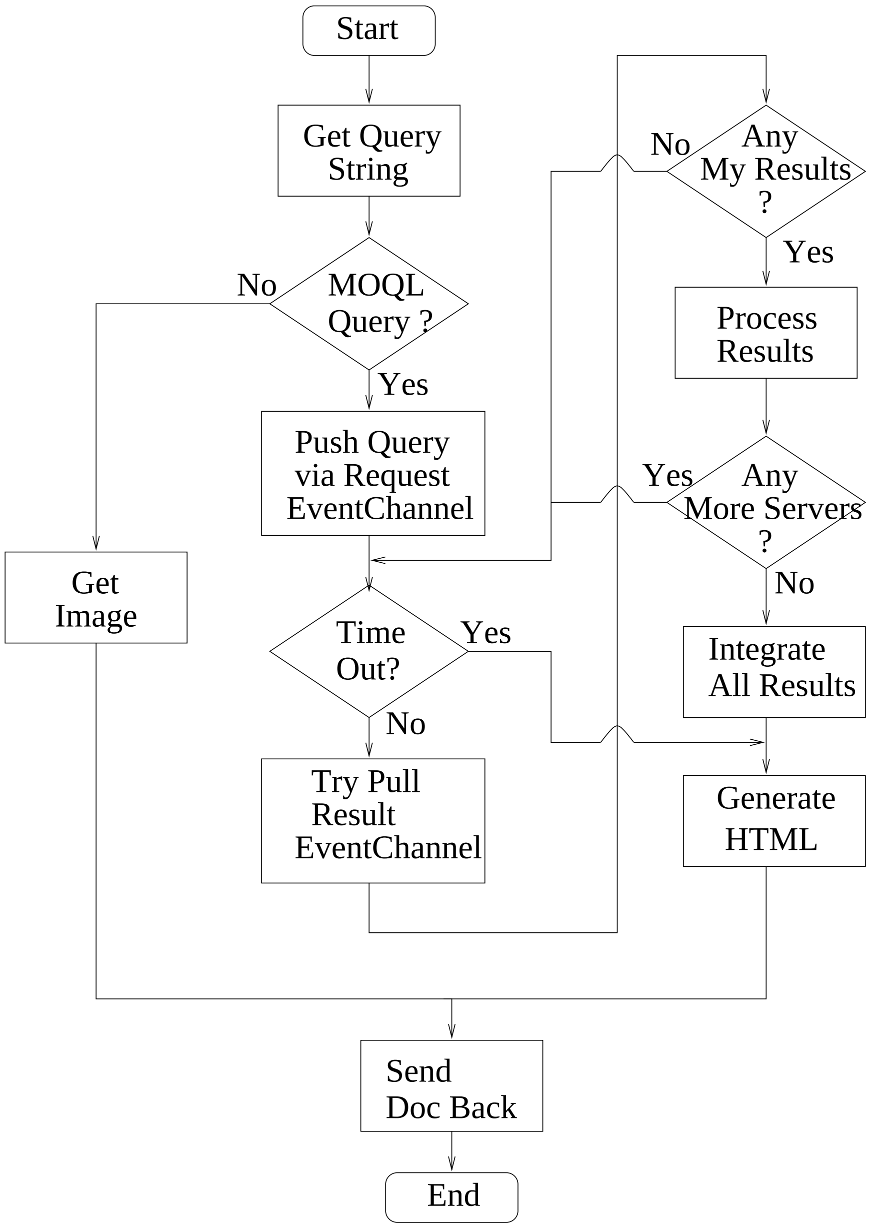
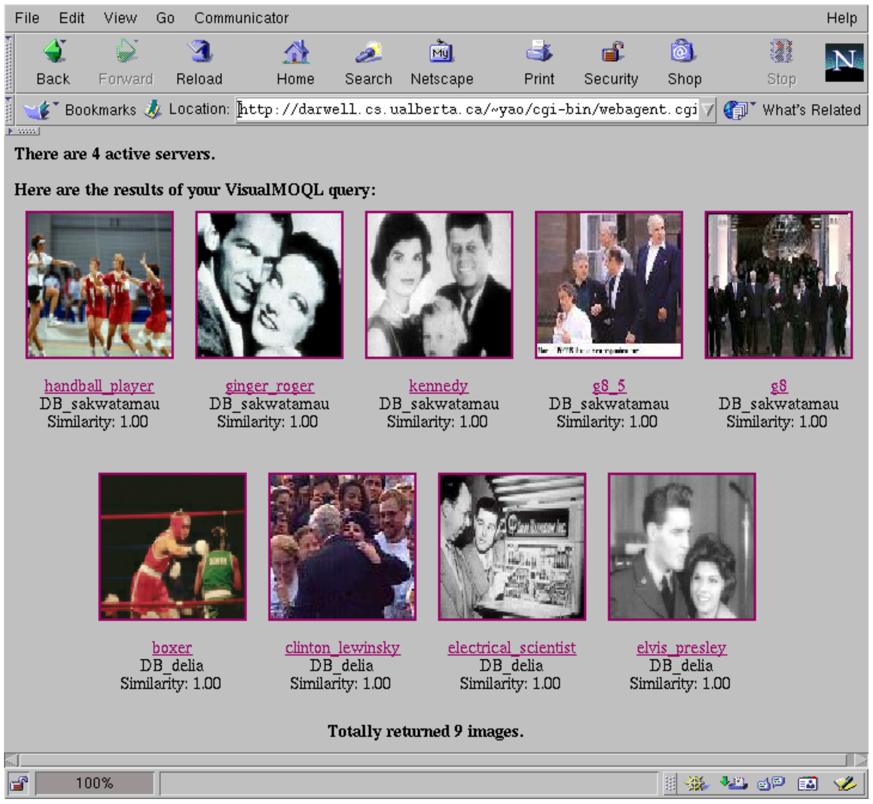


Figure 5.9: Flow Chart of Global Query Agent


File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://darwell.cs.ualberta.ca/~yao/cgi-bin/webagent.cgi
What's Related
100%

There are 4 active servers.

Here are the results of your VisualMOQL query:

handball_player
DB_sakwatamau
Similarity: 1.00

ginger_roger
DB_sakwatamau
Similarity: 1.00

kennedy
DB_sakwatamau
Similarity: 1.00

g8_5
DB_sakwatamau
Similarity: 1.00

g8
DB_sakwatamau
Similarity: 1.00

boxer
DB_delia
Similarity: 1.00

clinton_lewinsky
DB_delia
Similarity: 1.00

electrical_scientist
DB_delia
Similarity: 1.00

elvis_presley
DB_delia
Similarity: 1.00

Totally returned 9 images.

Figure 5.10: Distributed Query Results

Chapter 6

Conclusion and Future Work

In this technical report, we describe the design and implementation of building an interoperable distributed image database management system on top of a distributed object oriented computing platform, CORBA. Based on the object oriented methodology, the interoperable integration framework provides users with uniform interfaces for accessing images from multiple, disparate data repositories. We can take advantage of the distributed environment and sufficiently exploit image resources, while keeping the autonomy of each individual data repository.

Each image data repository participating in the framework is encapsulated as a DBMS-like CORBA object with a generic interface to the global query agent. This provides a single database illusion to users. The global query agent is responsible for passing queries from users to all available DBMS-like servers, integrating result images from each server matching queries, and sending the results back to users.

Clearly, CGI programming is the bottleneck of our distributed system. A possible improvement is that using *Java Servlets* [22] as a replacement for CGI programs. A Java servlet is a Java class that can be loaded dynamically to expand the functionality of a server (mostly a Web server), and it takes the place of CGI scripts. A servlet runs inside a Java Virtual Machine on the server, so it is safe and portable. One important feature of servlets is that different programs and/or requests are all handled by separate threads within one main web server process, while CGI uses multiple processes to handle separate programs and/or separate requests. By using the Java servlet programming, we can achieve the same functionalities of the CGI programming, without the difficulties of CGI.

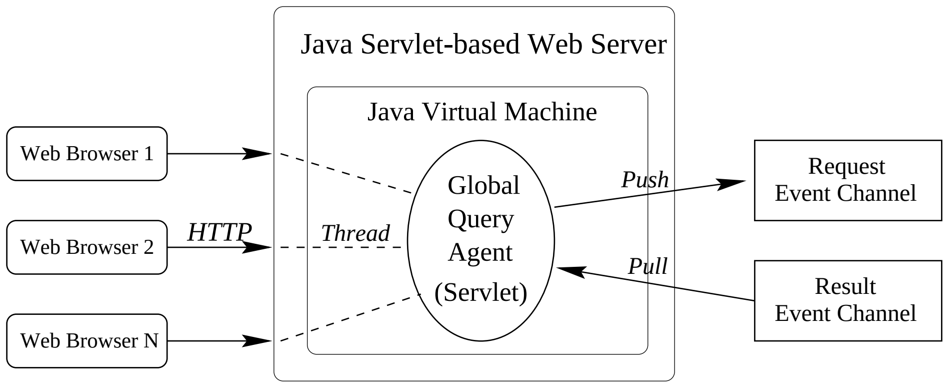


Figure 6.1: Introduce Java Servlet into the DISIMA Architecture

In Figure 6.1, we introduce the Java servlet programming into DISIMA architecture. The Global Query Agent is a servlet, as well as a CORBA object but, throughout its life cycle, it only has one instance. When a user first accesses the global query agent through the Web server, that user is assigned a new `HttpSession` object and a unique session Id, which can be accessed by the global query agent using the `getId()` method:

$$public\ String\ HttpSession.getId()$$

Thus, the global query agent can use this Id as a unique tag for each event so that each user can get exactly what (s)he wants.

We need to implement the global query agent using the Java language so that it can fit into a Java virtual machine. Since MICO ORB only supports C/C++ language, we can use *JacORB*, a free implementation of the CORBA standard supporting Java. Slight changes need to be made for the VisualMOQL's interface to the global query agent.

In Section 5.5, we mentioned three implementation issues — CGI, Image Integration and IP Address. By using Java servlet, we will efficiently solve the first and third issues. The second issue needs to be further studied.

Additional further work is to efficiently resolve the database model and semantic heterogeneity in the distributed DBMS environment. Topics needing to be investigated include: how to map the global query language into a local query language (say from MOQL to OQL), how to translate the global data model into local data models, how to resolve schema conflicts between global and local schemas, and how to homogenize the meanings of the object.

Bibliography

[1] A. Dogac, C. Dengi, and M. T. Özsu. Building Interoperable Databases on Distributed Object Management Platforms. *Communications of ACM*, 41(9):95–103, September 1998.

[2] S. Allamaraju. Nuts and Bolts of Transaction Processing. URL: http://www.subrahmanyam.com/articles/transactions/NutsAndBoltsOfTP.html, July 1999.

[3] BEA Systems, Inc. URL: http://www.beasys.com.

[4] A. D. Bimbo. *Visual Information Retrieval*. Morgan Kaufmann Publishers, Inc., 1999.

[5] T. Boutell. *CGI Programming in C & Perl*. Addison Wesley Longman, Inc., 1996.

[6] R. Cattell. *The Object Database Standard: ODMG-93 (Release 1.1)*. Morgan Kaufmann, San Francisco, CA, 1994.

[7] I. L. Cheng. Image Databases: A Content-Based Type System and Query By Similarity Match. Master's thesis, Department of Computing Science, University of Alberta, May 1999.

[8] Hewlett-Packard Company. URL: http://www.hp.com/ovc/index.html.

[9] The World Wide Web Consortium. CGI: Common Gateway Interface. URL: http://www.w3.org/CGI/Overview.html.

[10] The World Wide Web Consortium. HTTP: Hypertext Transfer Protocol Overview. URL: http://www.w3.org/Protocols/Overview.html.

[11] The World Wide Web Consortium. HyperText Markup Language Home Page. URL: http://www.w3.org/MarkUp/Overview.html.

[12] META Group Consulting. CORBA vs. DCOM: Solutions for the Enterprise. URL: http://www.sun.com/swdevelopment/news/CORBA.shtml, March 1998. Sun Microsystems, Inc.

[13] Expersoft Corporation. URL: http://www.expersoft.com.

[14] IBM Corporation. URL: http://www.ibm.com/software/data/db2.

[15] IBM Corporation. URL: http://www.ibm.com/software/webservers/appserv.

[16] Informix Corporation. URL: http://www.informix.com/informix/products/servers.

[17] Microsoft Corporation. URL: http://www.microsoft.com/sql/default.htm.

[18] Microsoft Corporation. URL: http://www.microsoft.com/NTServer/appservice/exec/overview/dcombus.asp.

[19] Oracle Corporation. URL: http://www.oracle.com/database.

[20] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, second edition, 1994.

[21] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley Longman, Inc., 1999.

[22] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly & Associates, Inc., 1998.

[23] J. Z. Li, M. T. Özsu, D. Szafron, and V. Oria. MOQL: A Multimedia Object Query Language. In *3rd International Workshop on Multimedia Information Systems*, pages 19–28, Como, Italy, September 1997.

[24] M. T. Özsu, X. Li, and L. Liu. DISIMA - A Distributed Image Database Management System. URL: http://www.cs.ualberta.ca/~database/research/ImageDB/proposal/proposal.html, November 1995.

[25] Object Design, Inc. *ObjectStore ObjectForms User Guide*, July 1997. Release 2.0.

[26] Object Management Group, Inc. Common Facilities Architecture. Revision 4.0, November 1995.

[27] Object Management Group, Inc. A Discussion of the Object Management Architecture. January 1997.

[28] Object Management Group, Inc. CORBA Messaging. URL: http://www.omg.org/cgi-bin/doc?orbos/98-05-05, May 1998.

[29] Object Management Group, Inc. CORBA/Firewall Security. URL: http://www.omg.org/cgi-bin/doc?orbos/98-05-04, May 1998.

[30] Object Management Group, Inc. CORBAservices: Common Object Services Specification. December 1998.

[31] Object Management Group, Inc. Interoperable Naming Service. URL: http://www.omg.org/cgi-bin/doc?orbos/98-10-11, October 1998.

[32] Object Management Group, Inc. minimumCORBA. URL: http://www.omg.org/cgi-bin/doc?orbos/98-08-04, July 1998.

[33] Object Management Group, Inc. CORBA Component Scripting. URL: http://www.omg.org/cgi-bin/doc?orbos/99-08-01, August 1999.

[34] Object Management Group, Inc. CORBA Components. URL: http://www.omg.org/cgi-bin/doc?orbos/99-02-05, March 1999.

[35] Object Management Group, Inc. Fault Tolerant CORBA. URL: http://www.omg.org/cgi-bin/doc?orbos/99-12-08, December 1999.

[36] Object Management Group, Inc. Real-Time CORBA. URL: http://www.omg.org/cgi-bin/doc?orbos/99-02-12, March 1999.

[37] Object Management Group, Inc. The Common Object Request Broker: Architecture and Specification. Minor Revision 2.3.1, October 1999.

[38] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Inc., second edition, 1999.

[39] D. C. Schmidt and S. Vinoski. Object Adapters: Concepts and Terminology. *SIGS C++ Report*, 9(11), November/December 1997.

[40] J. Siegel. What's Coming in CORBA 3. URL: http://www.omg.org/news/pr98/compnent.html, 1999. Object Management Group, Inc.

[41] Sun Microsystems, Inc. URL: http://www.sun.com/software/neo.

[42] Sybase, Inc. URL: http://www.sybase.com/products/databaseservers.

[43] IONA Technologies. URL: http://www.iona.com.

[44] URL: http://www.mico.org. *MICO Is CORBA: An Open Source CORBA 2.3 Implementation*, 1999. Version 2.3.0.

[45] V. Oria, B. Xu, and M. T. Özsu. VisualMOQL: A Visual Query Language for Image Database. In *4th IFIP 2.6 Working Conference on Visual Database Systems - VDB 4*, pages 186–191, L'Aquila, Italy, May 1998.

[46] V. Oria, M. T. Özsu, B. Xu, L. I. Cheng, and P. J. Iglinski. VisualMOQL: The DISIMA Visual Query Language. In *Proceedings of the 6th IEEE International Conference on Multimedia Computing and Systems*, volume 1, pages 536–542, Florence, Italy, June 1999.

[47] V. Oria, M. T. Özsu, D. Szafron, and P. J. Iglinski. Defining Views in an Image Database System. In *8th IFIP 2.6 Working Conference on Database Semantics (DS-8) "Semantic Issues in Multimedia Systems"*, pages 231–250, Rotorua, New Zealand, January 1999.

[48] V. Oria, M. T. Özsu, L. Liu, X. Li, J. Z. Li, Y. Niu, and P. J. Iglinski. Modeling Images for Content-Based Queries: The DISIMA Approach. In *2nd International Conference on Visual Information Systems*, pages 339–346, San Diego, CA, December 1997.

[49] B. Xu. A Visual Query Facility for DISIMA Image Database Management System. Master's thesis, Department of Computing Science, University of Alberta, April 2000.

[50] Z. Yang and K. Duddy. CORBA: A Platform for Distributed Object Computing. In *Operating Systems Review*, volume 30(2), pages 4–31. ACM SIGOPS, April 1996.

Appendix A

Glossary

AMI – Asynchronous Method Invocation

BOA – Basic Object Adapter

CERN – European Laboratory for Particle Physics

CGI – Common Gateway Interface

CLI – Call Level Interface

CORBA – Common Object Request Broker Architecture

DBMS – DataBase Management System

DCOM – Distributed Component Object Model

DDL – Data Definition Language

DII – Dynamic Invocation Interface

DISIMA – DIStributed Image database MAnagement system

DSI – Dynamic Skeleton Interface

EJBs – Enterprise Java Beans

ESIOP – Environment-Specific Inter-ORB Protocol

GIOP – General Inter-ORB Protocol

GIS – Geographical Information System

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

IDL – Interface Definition Langauge

IIOP – Internet Inter-ORB Protocol

IP – Internet Protocol

IR – Interface Repository

LSO – Logical Salient Object

MIS – Management Information System

MOQL – Multimedia Object Query Language

NCSA – National Center for Supercomputing Applications

NSERC – Natural Sciences and Engineering Research Council

ODMG – Object Data Management Group

OLE – Object Linking and Embedding

OMA – Object Management Architecture

OMG – Object Management Group

OODBMS – Object Oriented DataBase Management System

OQL – Object Query Language

ORB – Object Request Broker

POA – Portable Object Adapter

PSO – Physical Salient Object

QoS – Quality of Service

SSL – Secure Socket Layer

TCP/IP – Transmission Control Protocol/Internet Protocol

TII – Time Independent Invocation

URL – Uniform Resource Locator

WWW – World Wide Web