Building Component Database Systems Using CORBA

M. Tamer Özsu and Bin Yao
University of Alberta
Department of Computing Science
Edmonton,Alberta, Canada T6G 2H1
{ozsu,yao}@cs.ualberta.ca

April 12, 2000

1 Introduction

The "componentization" of database management system (DBMS) services is a topic of recent interest and discussion [SZ96]. The issue arises because of the divergent requirements imposed upon DBMSs by the new application domains that the technology has started to penetrate. Applications such as CAD/CAM, software development, office information systems, collaborative design, and multimedia information systems have widely varying requirements (see [Özs96], for example, for a discussion of the requirements of multimedia information systems). Building general purpose DBMSs that can address these varying demands requires either that the systems themselves are customizable, or that more open system architectures are developed that lend themselves to integration of different DBMSs with different requirements. In either case, componentization is a necessity, even though the granularity of components is different. In the former case, each component provides one DBMS function, leading to a customizable DBMS with "plug-and-play" components; in the latter case, each component is itself a DBMS, leading to an *interoperable* system.

Even though the first type of componentization has been discussed in literature, there are no known DBMS implementations that use componentization extensively (i.e., more than ordinary code modularization). However, there are many examples of interoperable DBMSs.

One of the fundamental requirements for building these systems is the availability of a proper infrastructure and a methodology. In this chapter, we start with the basic assumption that object technology provides the most promising methodology for building componentized DBMSs (of either type). Consequently, we argue that the distributed object platforms provide the necessary infrastructure for componentization. There are arguments that favor Internet-based approaches to building large scale interoperable systems. However, that

is beyond the focus of this chapter.

There are two distributed object computing platforms: Object Management Architecture from Object Management Group (OMG) and ActiveX from Microsoft. The former is now in its third iteration (commonly referred to as CORBA-3) and the latter started its life as COM/DCOM and has recently been renamed ActiveX. This chapter deals only with CORBA.

The Object Management Group was formed in 1989 as an industry consortium, with the aim of addressing the problems of developing interoperable, reusable, and portable distributed applications for heterogeneous systems, based on standard object oriented interfaces. It addresses this problem by introducing an architectural framework with supporting detailed interface specifications. OMG's role is that of an interface and functionality specifier; it does not develop software itself. In this chapter, we discuss the overall architectural framework that OMG has proposed (Section 2), the fundamental interoperability infrastructure called CORBA (Section 3), and the specifications for the other components of OMG's platform (Sections 4 and 5). Finally, we discuss some experiences with using CORBA infrastructure for database interoperability.

2 Object Management Architecture

The Object Management Architecture (OMA) [Obj97] is the architectural environment defined by (OMG) for facilitating interoperability. There are two related models in the OMA – the *object model* and the *reference model* – describing 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 computational model supported by OMA is client-server, and implementing peer-to-peer computing in this environment, though possible, is not easy.

OMA's 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 concepts related to object implementations, including methods, execution engines, and activation. The model is a generic one that provides objects, values, operations, types, classes, and subtype/supertype relationship among types. An object is an abstraction with a state and a set of operations. The operations have well-defined signatures, and the operations together with the signatures form the interface of each object. Each object and operation has a type. The communication between objects is by means of sending requests, whereby a request is an operation call with one or more parameters, any of which may identify an object (multi-targeting). The arguments and results are passed by value.

The reference model identifies and categorizes the components, interfaces,

and protocols that constitute the OMA. Figure 1 depicts four categories of components: *object services*, *common facilities*, *domain interfaces*, and *application interfaces*. These are linked by an *object request broker* (ORB) component, which enables transparent communication between clients and objects.

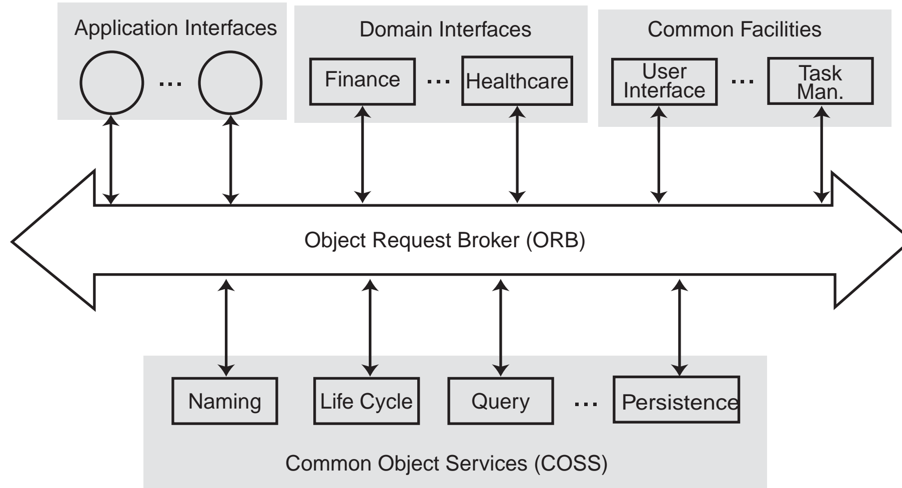

Figure 1: CORBA Architecture

The functions of each of these categories of components are described below. They will be discussed in more detail in the subsequent sections.

- The ORB component of the OMA is the communications infrastructure of the architecture, with a standard programming interface [Obj99a]. The interface is used by both the clients and the server objects. ORB allows objects to communicate transparently in a distributed environment, independent of the specific platforms. Its objective is to provide portability and interoperability of objects over a heterogeneous distributed system.

  The standard interface is provided by the *Interface Definition Language* (IDL). The IDL is a strongly typed declarative language that is programming language-independent. Language mappings from a variety of languages (currently, mappings have been specified for Ada, C, C++, COBOL, Java or Smalltalk) enable applications to be written in any of these languages, and access CORBA objects.

- Object services [Obj98b] are, in a sense, "value-added" services to facilitate the development of distributed applications. They are general purpose services that are domain independent and have standard IDL interfaces. Thus, they can be accessed by any client application just like any other CORBA object. A number of these services have been specified, including life cycle, naming, events, persistence, transactions, concurrency control,

relationships, externalization, licensing, query, properties, security, time, collections, and trading. We will discuss the details of these services in Section 4. They provide object abstractions and operations that are used by the other three classes of CORBA components. In theory, it is possible to put together a component DBMS by "gluing" together various CORBA services.

- Common facilities [Obj95] are interfaces for common (i.e., horizontal) facilities that are applicable to most application domains, and may be used by many applications. There are four major collections of common facilities: user interface, information management, system management, and task management. The facilities follow the fundamental OMA principle of providing their operational interfaces via the common IDL language. We discuss facilities in more detail in Section 5.

- Domain Interfaces are application domain-specific (i.e., vertical) interfaces that provide abstractions for various application domains. These abstractions are developed by OMG-sponsored industry groups. The ones that have received significant attention so far are finance, health care, manufacturing, telecommunication, electronic commerce, and transportation. In earlier versions of the CORBA specification, these were identified as Vertical Common Facilities.

- Application interfaces are external application interfaces that users develop. These applications do not have to be constructed using an object-oriented pattern or in an object-oriented language. Non-object-oriented applications can be "wrapped" in objects, and participate in a distributed OMA application in that manner.

3 Common Object Request Broker – CORBA

CORBA (Figure 2) is the key communication mechanism of OMA, in which objects communicate with each other via an Object Request Broker (ORB) that provides brokering services between clients and servers. Brokering involves target object location, message delivery, and method binding. Clients send a request to the ORB asking for certain services to be performed by whichever server can fulfill those needs. ORB finds the server, passes it the message from the client, and receives the result, which it then passes to the client.

The ORB performs the following functions, which we will describe in some detail below:

- Request dispatch to determine the identity of the callee method.

- Parameter encoding to convey local representation of parameter values.

- Delivery of request and result messages to the proper objects (which may be at different sites).

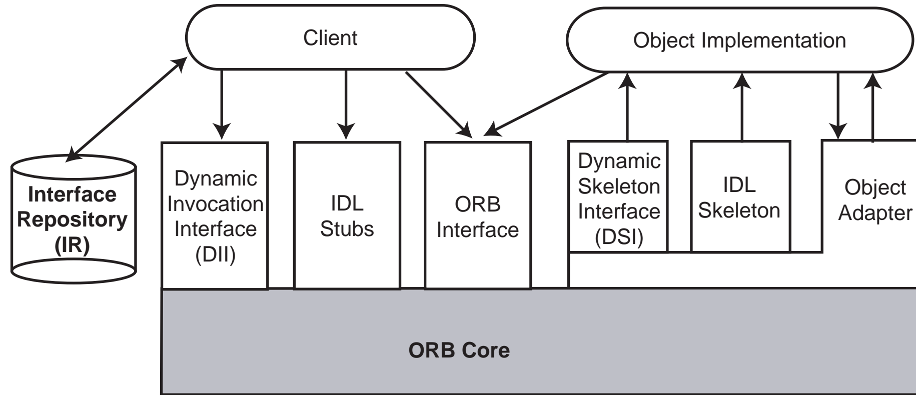

Figure 2: CORBA Architecture

- Synchronization of the requesters with the responses of the requests.
- Activation and deactivation of persistent objects.
- Exception handling to report various failures to requesters and servers.
- Security mechanisms to assure the secure conveyance of messages among objects.

The clients access the server objects by means of their interfaces. The interfaces are defined by means of the Interface Definition Language (IDL). For each CORBA object that is registered to the system, an IDL specification of its interface is necessary. IDL is a host language-independent, declarative language, not a programming language. It forces interfaces to be defined separately from object implementations. Objects can be constructed using different programming languages and still communicate with one another. IDL enables a particular object implementation to introduce itself to potential clients by "advertising" its interface. The IDL interface specifications are compiled to declarations in the programmer's own language. Language mappings determine how IDL features are mapped to facilities of a given programming language. There are standardized language mappings for C, C++, Smalltalk, Ada and Java. IDL language mappings are those in which the abstractions and concepts specified in CORBA meet the "real world" of implementation. Because of its central role, we first discuss IDL in some detail.

3.1 Interface Definition Language

IDL defines the types of CORBA objects by specifying their interfaces. An object's interface is composed of a set of client accessible operations and the arguments to those operations. IDL is purely a declarative language, and, therefore,

clients and servers can not be implemented directly in IDL. The only purpose of the IDL is make interfaces to be defined in a language independent way, which allows applications developed in different programming language to interoperate.

The OMG IDL obeys the same lexical rules as C++, but adds some new keywords (e.g. *any, attribute, interface, module, sequence, and oneway*) to support distribution concepts. The IDL grammar is a subset of the proposed ANSI C++ standard (supports only syntax for constant, type, and operation declaration, not any algorithmic structures or variables), with additional constructs to support the operation invocation mechanism. The restrictions imposed by IDL are the following:

- 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 IDL interfaces is that they can inherit from one or more other interfaces. This feature allows new interfaces to be defined from 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 following example that defines the interface through which the client can get sets of images from a image DBMS.

```
1  //Example 1 - Filename ‘‘queryagent.idl’’
2
3  module DB {
4
5    interface QueryAgent {
6
7      //exception
8      exception SyntaxError {
9        unsigned short position;
10       string errMessage;
11     };
12
```

```
13      //user defined types
14      typedef sequence<octet> streamType;
15
16      //content of an image
17      struct imageType {
18        string serverID;
19        string imageID;
20        string imageLabel;
21        streamType imageStream;
22        float similarity;
23      };
24
25      typedef sequence<imageType> imagesType;
26
27      //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 the 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 is about interfaces to bank account transactions. It covers most of the features of the IDL, although some operations are not realistic. This example also demonstrates the substitutability principle discussed above. 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.

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

```
3  //establish a unique prefix for interface repository IDs
4  #pragma prefix "bank.com"
5
6  module BANK {
7
8    //types
9    enum AccountType {CHECKING, SAVING};
10   //constant
11   const unsigned long MAX_LENGTH = 20;
12   typedef sequence<char, MAX_LENGTH> AccountNum;
13
14   interface Account {
15     //attributes
16     readonly attribute AccountNum check_account_num;
17     readonly attribute float check_account_balance;
18     readonly attribute AccountNum save_account_num;
19     readonly attribute float save_account_balance;
20     readonly attribute string pin;
21              attribute string address;
22   };//end of interface Account
23
24   //interface inheritance:
25   //AccountTransaction inherits attributes of Account
26   interface AccountTransaction : Account {
27
28     //exceptions
29     exception account_invalid {
30       string reason;
31     };
32     exception incorrect_pin {};
33
34     //operations and raising exceptions
35     float balance (in AccountType account_type,
36                    in AccountNum account_num,
37                    in string pin)
38             raises (account_invalid, incorrect_pin);
39
40     void deposit (in AccountType account_type,
41                   in AccountNum account_num,
42                   in float amount,
43                   out float new_balance)
44            raises (account_invalid);
45
46     //one-way
47     oneway void withdraw (in AccountType account_type,
48                           in AccountNum account_num,
```

```
49                            in float amout,
50                            in string pin);
51
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 defines an interface *Account* which includes the primary information of a bank account, such as pin number, home address, account numbers, account balancesc. 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 can not return any values, nor can it raise exceptions.

As indicated above, multiple interfaces are supported by means of inheritance. The newly proposed CORBA-3 standard extends this functionality by allowing a composite object to manage independent interfaces that are not supported via inheritance.

3.2 Client-CORBA Object Communication

The basic communication between two CORBA objects is accomplished by the ORB core. OMG does not place any restriction on how ORBs are implemented. In most ORB implementations, existing IPC (Inter-Process Communication) methods, such as Unix socket libraries, shared memory and multithreaded libraries, are used to achieve actual communication among clients, servers, and

the ORB. Yet the ORB can be as simple as a library that supports communication among objects and their clients, which are co-resident in the same process space.

To make a request, the client needs to know the operations that it is going to request of an object. In other words, it needs to know the interface of the object that will respond to the request. From here, the client can determine the reference of the target object which will service the request. This can be obtained either from the ORB as the reference of an existing object that is generally created by an object factory, or by using the Naming Services – one of the COSS modules. The target object reference, together with the requested operations, constitutes the request.

Clients can communicate (or access) CORBA objects in one of two ways. One way, known as *static invocation*, establishes the linkage between the client and the server at compile time. To facilitate this linkage, IDL compilers generate client-side *stubs* and server-side *skeletons*. These are interface-specific code segments that cooperate to effectively exchange requests and results. A stub is a mechanism that creates and issues requests on the client's behalf. A skeleton is a mechanism that delivers requests to CORBA object implementation.

An alternative is to use *dynamic invocation* through the Dynamic Invocation Interface (DII) at the client side, and the Dynamic Skeleton Interface (DSI) at the server side. DII allows clients to send requests to servers without the compile-time generation of stubs; DSI allows servers to be written without skeletons. Applications that establish static invocation bindings at compile time execute faster and are easier to program, since the programming interface is similar to ordinary object-oriented programs – a method is invoked on an identified object. Furthermore, static invocation permits static type checking, and the code is self-documenting. However, dynamic invocation is more flexible, leading to code genericity, and it allows runtime addition of CORBA objects and classes, which are needed for various tools, such as schema browsers.

For either of these modes, but particularly for dynamic invocation, the client application must have a way to know the types of interfaces supported by the server objects. The CORBA Interface Repository (IR) allows the IDL type system to be accessed and written programmatically at run time. IR is itself a CORBA object that has a standard interface. Using this interface, an application can traverse the entire hierarchy of IDL information.

To use or implement an interface, the language-independent interface defined in the IDL must be mapped (using an IDL compiler) into corresponding type 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. The generic mapping process is shown in Figure 3, although there are differences between various implemented ORBs, as well as for different languages.

IDL definitions of the interfaces of CORBA objects are precompiled to generate skeleton sources (which are server object templates without implementation) and client stub sources. At this stage, the interface definition is also inserted into the IR. The skeleton sources are compiled together with the implementation

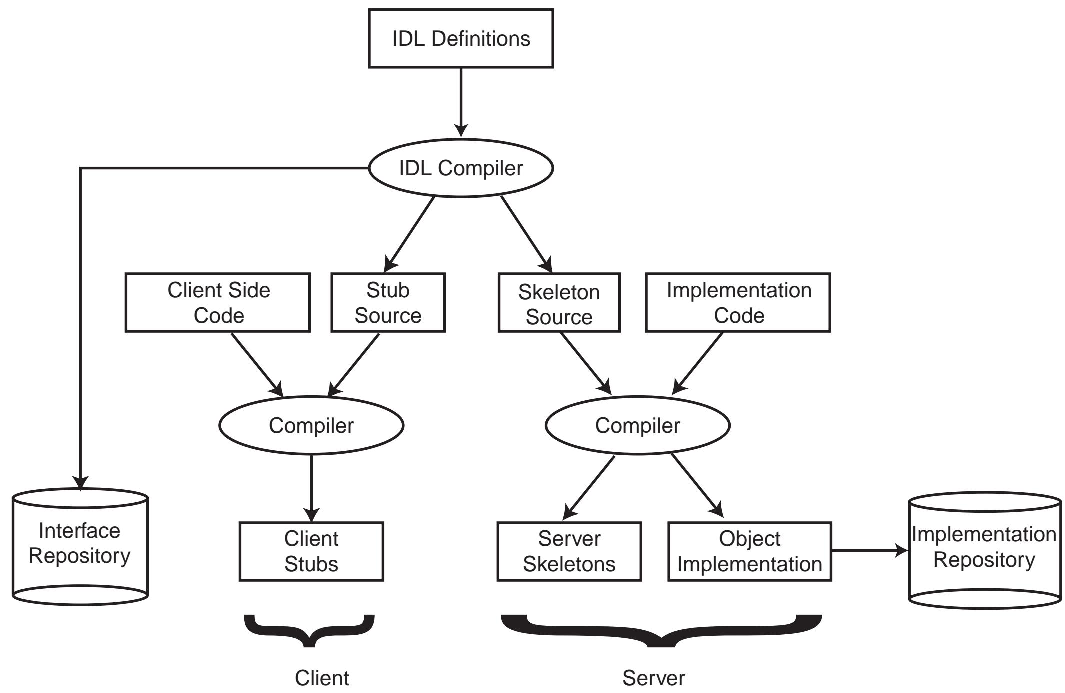

Figure 3: IDL Processing

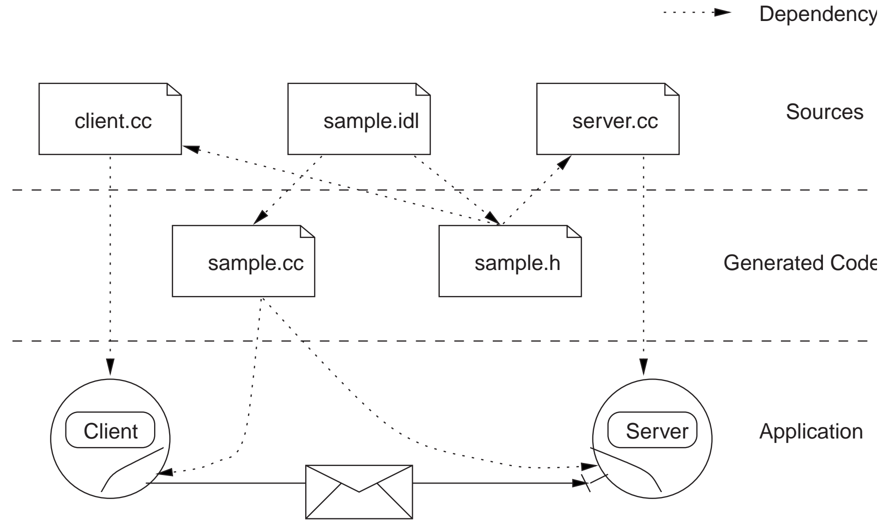

Figure 4: Creating a CORBA application

code, to implement the interfaces defined by IDL to generate server side skeletons (this is the OMG term for server side stub) and the object implementation code. The latter is inserted into the Implementation Repository to keep track of the code. The client stub sources are compiled together with the client side code to generate client stubs. The two compilers need not be the same and are dependent upon the language in which the server object and client application are written.

Figure 4 shows the creation process of a client and a server in C++, as implemented in the MICO ORB. 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 a 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 a executable CORBA server.

Based on the flow of requests from the client to the server, via the ORB, the following steps occur:

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 (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 3.4).

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 a 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.

A second issue in client-server communication is the association mode between a client request and a server method. CORBA provides three alternatives for this: one interface to one implementation, one interface to one of many implementations, and one interface to multiple implementations. If there exists only one implementation of an interface, all of the requests should be directed to a server that supports this single implementation. If there is more than one implementation of an interface, ORB can direct the requests to a server that supports any one of the existing implementations. In both cases, implementations handle all operations defined in the interface, and after implementation selection, ORB always uses the same implementation for requests to a particular object. If each implementation of an interface does not handle all of the operations defined in the interface–that is, if each implementation provides only a part of the interface–the third method is used for associating a client request

with a server method. In this case, ORB directs the requests to a server that supports an implementation of the interface that handles the invoked operation.

Finally, there is the issue of the call communication modes. CORBA versions prior to CORBA-3 define three modes between a client and a server – namely, synchronous, deferred synchronous, and one-way. Synchronous mode refers to blocked communication, where the client waits for the completion of the requested operation. Synchronous mode can be restrictive for clients who issue operations that can be executed in parallel with multiple objects. In deferred synchronous mode, the client continues its execution after server selection and keeps polling the server to get the results until the operation is completed. In one-way operation, a client sends a request without any intention of getting a reply.

CORBA 2.1 does not support asynchronous mode, which implies that if a client is to receive asynchronous messages, it should also act as a server that implements an object that can receive requests. In other words, the asynchronous mode of operation can be achieved between two CORBA objects by sending one-way requests to each other. The only disadvantage of this peer-to-peer approach is the increased complexity of the client code.

With CORBA-3, asynchronous messaging is introduced. In fact, the new standard [Obj98a] introduces two new invocation modes: *asynchronous method invocation* (AMI) and *time independent invocation* (TII), both of which can be used in static and dynamic invocations. 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.

3.3 Object Adapters

Object implementations access most of the services provided by the ORB via object adapters (OA). Each OA is an interface to the ORB allowing it to locate, activate, and invoke operations on an ORB object. The OA is a "glue" between CORBA object implementation and the ORB itself, adapting the interface of another object to the interface expected by a caller. OA uses delegation to allow a caller to invoke requests on an object, even though the caller does not know the object's true interface. The fundamental functions that an object adapter performs are the following:

- 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, and

- Registration of implementations.

Until recently (i.e., until CORBA 2.1), only the Basic Object Adapter (BOA) was defined and had to be provided by all commercial ORBs. BOA is designed to be used with most object implementations and provides for generation and interpretation of object references, method invocation, registration, activation and deactivation of object implementations, selection of proper object implementation for a given object reference, and authentication.

Included with CORBA 2.2, OMG released a an alternative to BOA. This standard, called the Persistent Object Adapter (POA), provides ORB portability. Some CORBA products have already started to include POA as part of their basic system offering. There are expectations that, in the future, OMG will publish another standard for object DBMSs. Since object DBMSs provide some "ORB-like" services, such as object reference generation and management, this adapter will be tuned to integrate object DBMSs with ORB distribution and communication. Library object adapters will be tuned for implementations resident in the client's process space. With the introduction of POA, OMG has removed the BOA specification from CORBA, even though many ORBs continue to support it. We will discuss POA and its functionality in more detail.

POA is responsible for the entire life-cycle of a CORBA object – from its creation to its destruction. 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 *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 [Hoq98]. 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 [HV99]. 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 5 shows the abstract POA model while a request sent from the client is dispatched to a servant. The process is as follows:

1. The server exports an object reference for an object. Only as a result of this export does the object become known to the sytem.

2. A client accesses this object by its object reference. The client may obtain this object reference in a number of ways; OMA provides two services (see Section 4) that can be used: the *naming service* and the *trader service*.

3. The client ORB uses the object reference to dispatch the request to the server ORB.

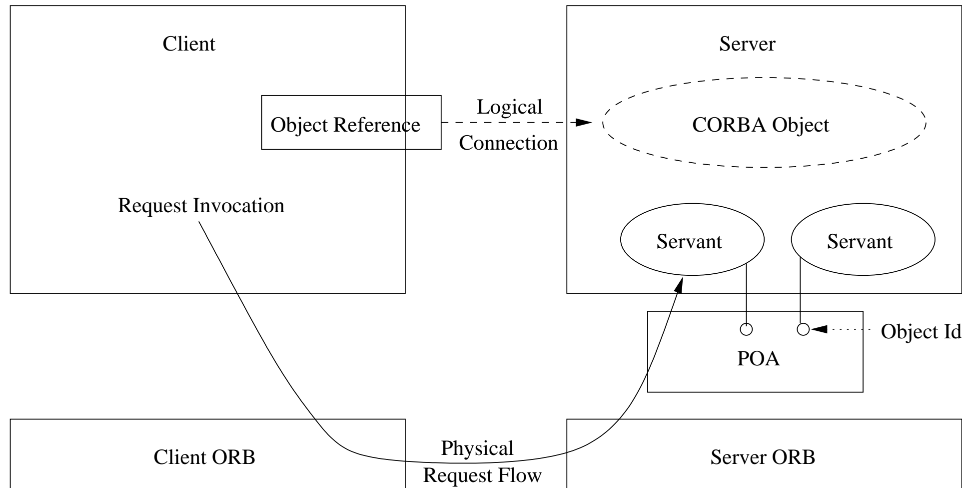

Figure 5: Abstract POA Model (Adapted from [HV99])

4. The server ORB then dispatches the request to the Object Adapter.
5. The Object Adapter dispatches the request to the appropriate servant (identified by Object ID) that incarnates the target object.

In the case of POA, the above process is slightly modified. This is due to a feature of POA which allows a server to have more than one POA Thus, in Step 4, the ORB has to find and dispatch the request to the particular POA instance that services the server object. In the case of multiple POAs, each one 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. POAs can be defined as specializations of other POAs from which they inherit policy values. The POA policies are described below:

- *Thread Policy:* A POA can either have the ORB control its threads or be single-threaded.

- *Lifespan Policy:* This policy is to specify whether the objects created within a POA are transient or persistent. Transience is defined with respect to the lifetime of the server process in which the object is created.

- *Object ID Uniqueness Policy:* The decision here is whether a servant can be associated with only a single object or with multiple objects. The servants associated multiple objects can reduce the server's memory use.

- *ID Assignment Policy:* The policy means objects created with that POA are assigned Object IDs only by the POA; otherwise, Objects IDs are assigned only by the server.

- *Servant Retention Policy:* Can a POA retain the associations between servants and objects across request boundaries, or does it establish a new association for each incoming request?

- *Request Processing Policy:* When a request arrives for a specific object, the POA can act as follows:
  - If the Object ID is not among the currently active objects, an exception is returned to the client.
  - If the Object ID is not among the currently active objects or the non-retention policy is chosen for the servant, and a default servant has been registered with the POA, that servant can be used to service the request.
  - If a *servant manager* has been registered with the POA, it is invoked by the POA to locate a servant or raise an exception.

- *Implicit Activation Policy:* If the implicit activation policy is chosen, a POA can activate a servant implicitly. This is useful for registering servants for transient objects.

When registering objects to CORBA, it is necessary to specify an activation policy for the implementation of each kind of object that will be used by the object adapter. This policy identifies how each implementation gets started. An implementation may support shared, unshared, server-per-method or persistent activation policies. While a server that uses a shared activation policy can support more than one object, a server that uses an unshared activation policy can support only one object at a time for an implementation. In the server-per-method activation policy, a new server is used for each method invocation. The persistent activation policy is similar to the shared activation policy, except that the server is never started automatically.

3.4 ORB Interoprability

Earlier versions of CORBA (prior to Version 2.0) suffered from a lack of interoperability among various CORBA products, caused by the fact that earlier CORBA specification did not mandate any particular data formats or protocols for ORB communications. CORBA 2.0 specifies an interoperability architecture based on the *General Inter-ORB Protocol* (GIOP), which specifies transfer syntax and a standard set of message formats for ORB interoperation over any

connection-oriented transport. CORBA 2.0 also mandates the *Internet Inter-ORB Protocol* (IIOP), which is an implementation of GIOP over TCP/IP transport. With IIOP, ORBs can interoperate with one another over the Internet.

4 Common Object Services

Object Services provide the main functions for implementing basic object functionality using ORB. Each object service has a well-defined interface definition and functional semantics that are orthogonal to other services. This orthogonality allows objects to use several object services at the same time without any confusion.

The set of services are diverse and at different phases of development. Some of these services are fundamental to any distributed application development over CORBA: *Naming Service, Event Service, Life Cycle Service,* and *Persistent Object Service.* Some others are also important for development of component DBMSs. Some of these are important database-related object services, including *Transaction Service, Concurrency Control Service*, and *Query Service.* Other services defined by OMG include: *Relationship Service, Externalization Service, Licensing Service, Property Service, Security Service, Time Service, Collections Service*, and *Trading Service.* Not only are there differences among these with respect to their level of development as a standard, but there are differences in the development of commercial services by each vendor. We discuss these services in this section, giving more emphasis to the fundamental ones and to those that are important for component DBMSs.

The provision of these services, and their use by other CORBA objects, provide "plug-and-play" reusability to these objects. As an example, a client can move any object that supports Lifecycle Services by using the standard interface. If the object does not support the standard Lifecycle Services, then the user needs to know "move semantics" for the object and its corresponding interface.

4.1 Naming Service

The Naming Service supports a name-to-object association called a *name binding.* The binding of a name to an object is done relative to a *naming context*, which 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. Resolution of a name is the determination of the object associated with the name in a given context.

A context is like any other object, and it can be bound to a CORBA object or another context object. This results in a hierarchy of contexts and bindings known as a *naming graph* (Figure 6), which can be supported in a distributed, federated fashion. It is possible to view a naming graph as similar to a file

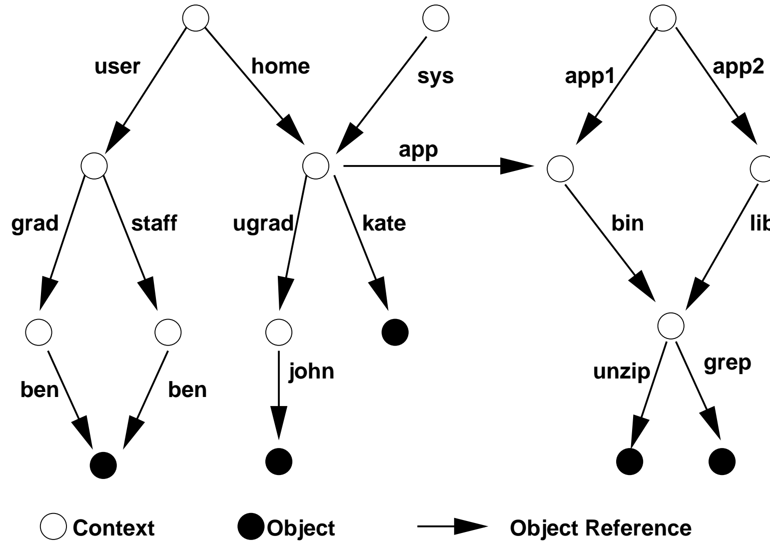

Figure 6: A Naming Graph

system, in which contexts are analogous to directories that store bindings to either directories (other contexts) or files (objects).

4.2 Event Service

The Event Service supports asynchronous events by decoupling the communication between objects. It allows objects to be invoked when certain events occur. An object can assume either a *supplier* role if it produces event data, or a *consumer* role if it processes event data.

The communication of event data between suppliers and consumers is accomplished by issuing standard CORBA requests through appropriate event channel implementations. An event channel is an object that facilitates multiple suppliers to communicate to multiple consumers asynchronously. This communication can follow either the *push model* or the *pull model*. In the former, the supplier object transfers event data to the appropriate consumer when the corresponding event occurs; in the latter, the consumer periodically requests event data from a supplier.

4.3 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 `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.

4.4 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 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.

4.5 Transaction Service

The *Transaction Service* supports ACID transactions. Thus, distributed objects that use the Transaction Service interfaces to cooperate with each other provide the following well-known ACID properties:

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 produces consistent results.

Isolation - A transaction is isolated from other transactions. Thus, 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 effects are persistent and survive future system failures.

Transaction service implementations have to support *flat* transactions, and the standard specification has provisions for supporting *nested* transactions. This service depends on the *Concurrency Control Service* to enforce isolation, and the *Persistent Object Service* to enforce durability.

4.6 Concurrency Control Service

This service coordinates the concurrent access to shared objects (i.e., resources) by multiple clients so that the object remains in a consistent state. It enforces the well-known serializability criterion for accesses by both *transactional* and *non-transactional* clients.

The synchronization approach is based on locking, where a lock is associated with each object. Different lock modes are defined in order to provide flexible conflict resolution.

4.7 Query Service

The *Query Service* provides query operations on collections of objects, and may return collections of objects. The query operations include predicate-based declarative specifications. The standard does not specify a particular query language; queries can be specified by SQL variants that support object-oriented concepts (e.g., SQL3 or ODMG OQL) or any other type of language.

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 IDL interfaces. Thus, the query service supports nested queries of the well-known form.

4.8 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 use 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.

4.9 Other Object Services

The *Trading Service* facilitates objects collaboration. An object that is available to provide a service registers its availability with the service by describing the service and specifying its location. Objects can then inquire of the trader for any services they need. Thus, the trader matches the available services to the needs of objects.

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.) and then internalize it into a new object in the same or different process.

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

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

The *Property Service* is capable of dynamically associating 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 the current time, together with an error estimate associated with it, to synchronize clocks in distributed systems.

5 Common Facilities

Common facilities consist of components that provide services for the development of application objects in a CORBA environment. These are "horizontal" facilities that consist of those services that are used by all (or many) application objects. Examples of these facilities include user interfaces, systems management, and task management. We do not discuss these any further in this chapter.

6 Building Componentized Applications

One of the strengths of distributed computing platforms, such as OMA, is that they facilitate the development of large scale distributed applications. However, the facilities provided by the OMA is probably too low-level to accomplish this aim. It is certainly possible to develop systems using the CORBA infrastructure and the provided services; there are many examples of this. However, application development in this environment still requires writing a significant amount of "glue" code. Since these are generally proprietary systems, true re-usability at the application level is hard to achieve.

What is required is a value-added framework that allows the development of components and the use of these components in other applications. By components, we mean self-contained modules that perform a particular function (usually at a higher level of abstraction than an ordinary object). Components have clear interfaces that allow them to be plugged into other components. In this type of environment, distributed applications are developed by putting together many of these components and enabling them to cooperate with each other over a backplane. CORBA establishes this backplane, but more work is needed to facilitate this view of application development.

The above requirement – and shortcoming – was recognized quite early in the OMA development cycle. OLE provided the required functionality in Microsoft's DCOM/OLE environment; there are third party OLE providers that encapsulate applications to give them component features. Until recently, there was no comparable component framework for OMA. At one point, it was expected that OpenDoc would serve in this capacity, but with the demise of the OpenDoc initiative, the need for the development of component technology became acute. With the recent release of the CORBA-3 specification, this issue is addressed. Even though there are no commercial ORBs that are fully CORBA-3 compliant, products should be expected to come to the market in the next two to three years.

Within CORBA-3, OMG has defined two specifications relevant to component technology: CORBAcomponents [Obj99c] and CORBAscripting [Obj99b]. Within CORBA, components are modeled as objects (as most things are). Consequently, CORBAcomponents introduces a new meta-type called `Component` which supports multiple independent interfaces (as indicated earlier, pre-CORBA-3 ORBs only support multiple interfaces by inheritance) and a `CORBAcomponents container` provides infrastructure to navigate among them. Other components can then be defined as subtypes of the `Component` type, providing for the development of a complete component type system.

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 those of `CORBAservices`. As discussed above, this facilitates the development of CORBA middleware as part of the application development. The end result is expected to be easier system development, since the application developers only need to know components and their interfaces, rather than having to know CORBA in an intimate way.

There is also a provision to interoperate with Enterprise Java Beans (EJB), which allows CORBA components to present themselves as JavaBeans to Java programs or other tools in the JavaBeans environments.

Component Scripting [Obj99b] is a standard scripting language to wire all components together, by means of which users can easily modify or upgrade applications constructed using the language. The specification maps component assembly to a number of widely used scripting languages.

We conclude this section by providing sample code to demonstrate how one goes about putting together an application over CORBA. We should note that our focus is on the main steps and we, by necessity, eliminate many details.

Recall our first IDL example. The purpose of the interface `QueryAgent` is to use the method `getImages` to retrieve a set of images from an image database based on the given query. If we implement both the client and the server using C++, the simplified source codes would include the following:

The *Client*

- 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 image database server object. `serverName` is a string which is the name of the image database.

  ```
  // 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 );
  ...
  ```

- Invoke `getImage` method in synchronous mode to get the image data. `retImage` is a pointer that points to a set of images. `queryString` is a string that contains the specified query in a certain query language.

```
retImage = client->getImage (queryString);
...
```

The *Server*

The IDL wrapped image database server is registered with CORBA implementation repository. The ORB automatically delivers the request from the client to the server, specifically on a particular method `getImages`. The implementation codes for that method declared in the IDL file are included in a class called `QueryAgent_impl` as follows:

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

Inside the method, the query string is sent to the local database through its call-level interfaces, and results are composed and returned to the invoker.

In the `main()` we should include the following codes:

- Initialize ORB and root POA, and create the servant that provides the services to the client.

  ```
  // 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_varpoa = 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 the server 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 all the initialization, the POA manager is activated and the server waits for requests from the client.

```
mgr->activate();
...
orb->run();
...
```

7 CORBA and Database Interoperability

As an object-oriented distributed computing platform, OMA, and in particular CORBA, can be helpful for database interoperability. The fundamental contribution is in terms of managing heterogeneity and, to a lesser extent, managing autonomy. Heterogeneity in a distributed system can occur at the hardware and operating system (which we can jointly call platform) level, communication level, DBMS level, and semantic level. CORBA deals mainly with platform and communication heterogeneities. It also addresses DBMS heterogeneity by means of IDL interface definitions. However, the real problem of managing multiple DBMSs in the sense of a multidatabase system requires the development of a global layer that includes the global-level DBMS functionality. One issue with which CORBA is not useful is semantic heterogeneity.

Using CORBA as the infrastructure affects the upper layers of a multidatabase system, since CORBA and COSS together provide basic database functionality to manage distributed objects. The most important database-related services included in COSS are Transaction Services, Backup and Recovery Services, Concurrency Services, and Query Services. If these services 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 involved objects. For example, by using a Transaction Service, implementing a global transaction manager occurs by implementing the interfaces defined in the Transaction Service specification for the involved DBMSs.

In this section, we discuss some of the design issues that must be resolved to use CORBA for database interoperability. This discussion is based, to a large extent, on [DDÖ98].

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 needed to process (e.g., querying and transactional control) 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 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 another design issue. As specified earlier, CORBA provides three alternatives for this: one interface to one implementation, one interface to one of many implementations, and one interface to multiple implementations. The choice of the 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.

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 interopreable 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.

8 Conclusions

In this chapter we discussed the Object Management Architecture and its components as a possible platform for developing component DBMSs. We focused on the issues that are relevant to the subject matter of this book. More details on CORBA can be found in many books, such as [Sie96] and [Hoq98], as well as the OMG specifications cited in this chapter.

The term "component database" may mean many things; a classification is given in Chapter 1. It is clear that OMA can facilitate many of these componentization efforts. Building interoperable DBMSs on top of component DBMSs has been tried many times, and there are working systems. It should also be possible to put together a single DBMS by "gluing" together the relevant OMA services such as query, transaction, persistence, and concurrency control service. There are problems in accomplishing this (such as the inability of the current object adapters to efficiently deal with fine granularity objects), but these may well be overcome. The fundamental criticism of this approach has been the performance overhead that CORBA would produce. Even though this is likely to be a problem, there have been no studies that quantify the overhead.

If componentization of DBMSs (using any definition) is to become a reality, then there is clearly a need for distributed computing platforms such as CORBA. From its modest beginnings, OMA has made significant strides to provide the infrastructure for the building of componentized distributed applications – if components are sufficiently large. It is still unclear whether it can support finer granularity componentization.

References

[DDÖ98] A. Dogac, C. Dengi, and M. T. Özsu. Distributed object computing platforms. *Communications of the ACM*, 41(9):95–103, September 1998.

[Hoq98] R. Hoque. *CORBA 3*. IDG Books, 1998.

[HV99] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley Longman, Inc., 1999.

[Obj95] Object Management Group, Inc. Common facilities architecture. Revision 4.0, November 1995.

[Obj97] Object Management Group, Inc. A discussion of the object management architecture. January 1997.

[Obj98a] Object Management Group, Inc. Corba messaging. http://www.omg.org/cgi-bin/doc?orbos/98-05-05, May 1998.

[Obj98b] Object Management Group, Inc. Corbaservices: Common object services specification. December 1998.

[Obj99a] Object Management Group, Inc. The common object request broker: Architecture and specification. Minorrevision 2.3.1, October 1999.

[Obj99b] Object Management Group, Inc. Corba component scripting. http://www.omg.org/cgi-bin/doc?orbos/99-08-01, August 1999.

[Obj99c] Object Management Group, Inc. Corba components. http://www.omg.org/cgi-bin/doc?orbos/99-02-05, March 1999.

[Özs96] M.T. Özsu. Future of database systems: changing applications and technological developments. *ACM Computing Surveys*, 28(4es), December 1996. Available at http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a85-ozsu/.

[Sie96] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley, 1996.

[SZ96] A. Silberschatz and S. B. Zdonik. Strategic directions in database systems - breaking out of the box. *ACM Computing Surveys*, 28(4):764–778, December 1996.