CorbaScript and CorbaWeb: A Generic Object-Oriented Dynamic Environment upon CORBA

Philippe Merle, Christophe Gransart, Jean-Marc Geib
Laboratoire d'Informatique Fondamentale de Lille - URA CNRS 369
Université de Lille 1, Bât M3
59655 Villeneuve d'Ascq Cedex - France
Phone: (33) 20 43 47 21
Email: {merle, gransart, geib}@lifl.fr

Abstract

Traditionally, distributed object-oriented applications are based on a static approach where each application knows, at compilation time, all object types which could be used. This paper presents an approach to the construction of a new type of object-oriented application over large scale distributed systems that we call a "generic" application. These applications use objects dynamically without prior knowledge of the object types at compilation time. We present two tools based on this approach: CorbaScript, a generic object shell interpreter and CorbaWeb, a generic object browser based on WWW technology. These tools are implemented via the Dynamic Invocation Interface and the Interface Repository from CORBA.

1 Introduction

This work is part of the OSACA project of the computer science laboratory of the University of Lille. The name stands for "Open Software Architecture for Cooperative Applications". This project aims at building an open support environment for cooperative applications. These applications facilitate the work of related users in a shared environment. For us, Cooperative Environments are made up of client and/or server applications which cooperate over a large scale networked system.

During the lifetime of cooperative environments, the challenge is one of dynamism. To meet this, they have extra requirements such as:

- remote access to shared centralized or distributed servers which store a large amount of complex, multimedia and evolutionary data,
- various interaction modes between humans and computers due to the very large panel of roles in cooperative activities,
- the need for connections and disconnections of clients to be done dynamically,
- the need to handle evolution in data or behavior which may often arise following modifications in the cooperative environment, and
- the ability to handle new clients or servers that can be designed and added to the environment (for example, when a new role is needed for cooperation).

All these evolutions have to be dynamically achieved without rewriting or recompiling other applications involved in the cooperation.

From the system point of view, we need first a low level protocol which allows interactions between large scale distributed processes in a dynamic and a uniform way under heterogeneity. We have investigated the use of the Common Object Request Broker Architecture (CORBA [OMG 93]). This architecture, designed by the Object Management Group, is an open object-oriented support scheme for distributed application integration [Manola 93].

Then, at an upper level, we must include new functionalities dedicated to cooperative environments. This present work, "Generic Object-Oriented Dynamic Environment (GOODE)", aims at handling the dynamic issue of shared cooperative environments.

In the future, a CORBA compliant environment will support several million objects structured by several thousand object interfaces. These interfaces will form hundreds of frameworks: GUI, naming service, trader service, financial service are framework examples. Each framework could be implemented by servers suited to the required quality-of-services needs (e.g. persistent, fault-tolerant, multi-threaded objects). The system design complexity will be hidden by the object-oriented decomposition [Meyer 88], [Wegner 90]. But user's application design will become very complex because the applications have to know all object types they will use.

To avoid this complexity, we have studied a generic approach to the creation of very useful kinds of client object-oriented applications, e.g. shell interpreters, browsers, and so on. We argue that a powerful object environment requires generic tools to configure, administrate, test and browse object frameworks. These tools must not be statically linked to a set of object types, but they have to dynamically adapt to their object environment. They have to use any object types at runtime but may not know any such types at compilation time. Clearly this targets the issue of dynamism in complex environments like cooperative ones.

We investigate the use of the advanced dynamic invocation mechanism provided by CORBA: our applications adapt to objects and interfaces dynamically at runtime using the Interface Repository (IR, a CORBA component) and build requests with the Dynamic Invocation Interface (DII, another component). We have tested these generic applications on two distinct Object Request Brokers (ORB) [Orbix], [ORBeline].

The first section presents and discusses the OMG CORBA solution issues and the dynamic invocation mechanism. The next two sections introduce our Generic Object-Oriented Dynamic Environment (GOODE) through two tools. The first one - CorbaScript - is a generic object-oriented shell interpreter (section 3). The second one - CorbaWeb - is a generic object browser based on the Web. Its architecture is detailed in section 4. The last section presents perspectives on this new kind of generic toolset.

2 CORBA

2.1 A Federated Object-Oriented Approach

In the last quarter century, all computing technologies have been influenced by the object-oriented approach: design methods (e.g. OMT), databases (e.g. O2), component libraries, programming languages (e.g. C++, Eiffel, SmallTalk), distributed object-oriented systems (e.g. Guide, Spring). But all these object-oriented approaches are often incompatible and in most cases cannot easily interoperate.

Recognizing the urgent requirement for a standard in object-oriented application interoperability, the industry has formed the Object Management Group (OMG) consortium to create a unified distributed architecture to promote object technologies. In 1991, it proposed the Object Management Architecture (OMA) guide [OMG 92]. The OMA gives an overview of a federated distributed object system. The first OMG specification was the Common Object Request Broker Architecture (or CORBA) [OMG 93].

2.2 A Common Object-Oriented Architecture

CORBA specifies an object-oriented messaging facility for a distributed object environment: a mechanism allowing any object to invoke the services of any other object. Thus, CORBA provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems. CORBA

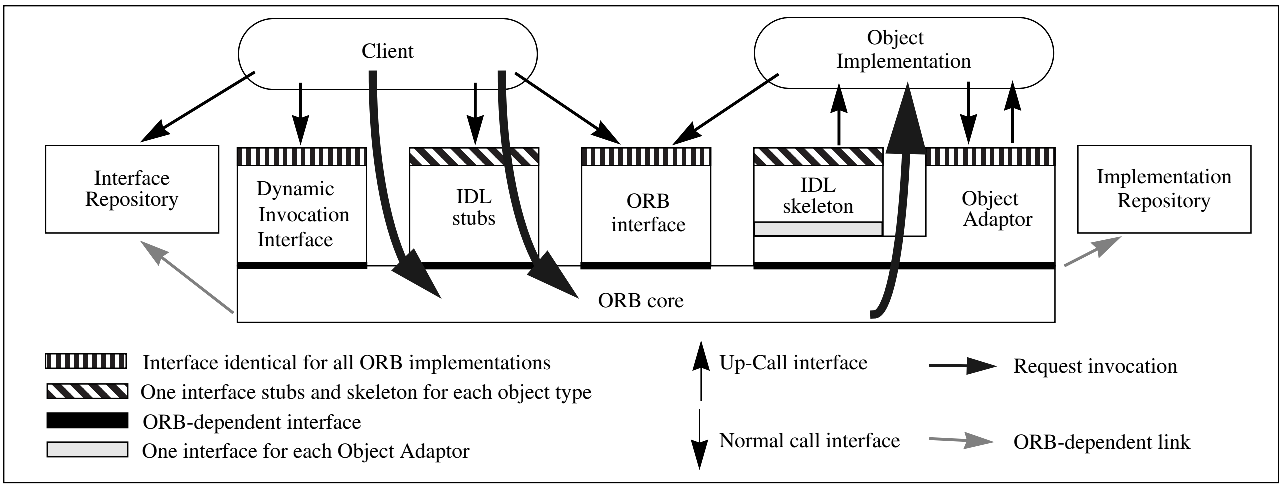

Fig.1: Common Object Request Broker Architecture

is specified independently of any programming language, since object interfaces are described in the Interface Description Language (IDL). A CORBA environment (see Fig.1) is decomposed into three main parts:

- **A client/server object-oriented invocation system**: The communication between objects is supported by a standard stubs mechanism generated by compilation of IDL descriptions. This separation hides distribution and permits heterogeneity between the service consumer (client) and the service producer (server).
- **An adaptive object support policy**: CORBA provides object adaptors to accommodate a diverse range of object features such as lifetime and granularity policies.
- **A reflexive environment**: CORBA provides at runtime objects to retrieve IDL information (Interface Repository). It facilitates dynamic type checking and allows the construction of requests through the Dynamic Invocation Interface API (described in section 2.4).

2.3 An IDL Banking Example

CORBA objects are specified by interface types described in the Interface Description Language. IDL is a strongly typed definition language which permits the definition of types, constants, exceptions, modules and interfaces. Each interface can contain attributes and operations with their type signature. IDL provides multiple interface inheritance to define new services reusing other service specifications. A complete description of IDL is given in [OMG 93].

```
interface account {
  readonly attribute float balance;
  void makeLodgement (in float f);
  void makeWithdrawal (in float f);
  struct description {
    string name
    float balance;
  };
  description describe ();
};
interface depositAccount : account {
  readonly attribute float interestRate;
};
interface bank {
  account newAccount (in string name);
  depositAccount newDepositAccount(
      in string name);
  struct description {
    string name;
    account account_object;
    float balance;
  };
  sequence<description> list_accounts ();
};
```

Fig.2: An IDL Banking Example

The IDL example (Fig.2) describes the interfaces for a banking service which contains three object types: `account`, `depositAccount` and `bank`. The `account` interface provides operations to update the balance (`makeLodgement`, `makeWithdrawal`) and functionalities to consult the state of objects (`balance`, `describe`). The `depositAccount` interface inherits from `account` and provides the new characteristic `interestRate`. The `bank` interface manages a set of accounts. It offers operations to create new accounts or deposit accounts and to obtain the list of accounts. IDL provides the functionality to construct complex structured types (`account::description`).

The IDL file may be compiled to generate stub and skeleton codes (see Fig.3). The skeleton code has to be integrated into a server program which will implement the various object types. The stub code has to be linked to client programs. Clients will invoke objects through the stub code. This is called the "static" approach and represents an object-oriented client/server methodology. But, if object types evolve or are added by the server designer, the client programs must be redesigned to use these new features. A more "dynamic" approach, introduced in the next sections, has to allow client programs to dynamically follow such evolutions.

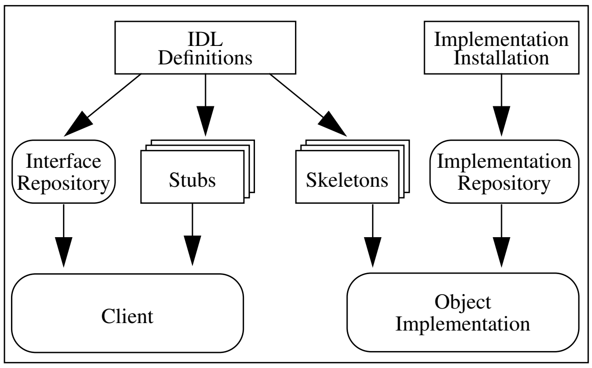

Fig.3: IDL Compilation

2.4 A Dynamic Approach

As shown before, CORBA has two invocation interfaces; the stub invocation mechanism is provided for client programs which know the object types at compilation time, and the Dynamic Invocation Interface (DII) is provided for clients which adapt to new object types at runtime. The Interface Repository is used here to retrieve this type information.

The Dynamic Invocation Interface is an API to construct requests at runtime. This API provides functions to fix the target object, the operation name and the argument list for a request. Each argument is composed of an argument name, a call mode (in, out, in-out or result), a type and a value. The types are represented at runtime by TypeCode class instances.

The Interface Repository is a structured object graph. Each node is an object which stores IDL informations. IR framework is composed of 20 interfaces (`ModuleDef`, `InterfaceDef`, ...).

When a program wants to invoke a newly discovered object, the general scheme to use the DII and IR is the following: it asks this object for its interface, the reply is an `InterfaceDef` object reference managed by the IR, then it queries the `InterfaceDef` object about the possible operations and attributes, it chooses which operation or attribute must be invoked, it constructs a request for the selected service, it invokes the request, it waits for the results, and finally it handles the results.

As we can see, using the DII mechanism is complex and has to be restricted to applications which specifically require this mechanism. Because of this complexity the DII mechanism is rarely used in actual CORBA developments. We argue that this mechanism is of very great interest in order to construct shells, browsers and configuration tools, which are very useful for dynamic distributed environments. Our plan is to create a generic environment based on dynamic invocation which will be used to manage the large scale distributed cooperative environment of OSACA.

To our knowledge, only few works explore this approach: TclDII [Jagannathan 94] , a TCL interpreter interfaced with the Orbix DII API; Web* a TclDII application which permits one to mix HTML pages with TclDII scripts. In the next section, we present our approach to the exploration of dynamic invocation issues.

3 CorbaScript

In this section, we present a first approach based on dynamic invocation. This allows us to simplify the application of methods on CORBA objects without initial knowledge of the object interfaces.

Today, the current applications and systems used with CORBA are composed of two or three hundred IDL interfaces: the Spring [Kessler 94] object-oriented operating system is composed of two hundred IDL interfaces, a classic ORB core with IR has 40 interfaces, a graphical user environment like Fresco [Fresco 94] is described by 30 interfaces. The Orbix Implementation Repository is made up of one object, but to handle this object 14 clients are needed (each client has approximately 500K of code for one operation). The naming service manages directories of objects. We can assume that exploitation of this service can be done with operations as in operating systems: touch, rm, mkdir ... These operations can be achieved by using specific clients (as for Orbix Implementation Repository) or using a shell interpreter which can call operations defined in the Implementation Repository IDL interface. In this section, we detail our shell proposal for CORBA.

3.1 An Object-Oriented Client/Server Model

CORBA is based on a distributed client/server object-oriented model. In practice, CORBA applications are composed of several client programs (user tools) and several server programs (information object managers). The server objects are accessible in a transparent way by client programs through the ORB. If we focus only on clients, we can highlight several classes of clients:

- A **specific client** is an application developed for a particular need i.e. a particular interaction with the concerned servers. This kind of applications is outside of the interest of this paper because such a client statically knows the object interfaces it will use and so the client application can be built upon the stub mechanism.
- A **navigational specific client** is an application which presents information to a user for a specific topic. For example, a file manager displays information about directories and files using a graphical user interface; this application can be seen as an encapsulation of shell functionalities. In the CORBA world, we can find navigational specific clients for Naming Service or Interface Repository with or without a graphical user interface.
- A **navigational general purpose client**: The best examples of navigational general purpose clients are Web clients. These applications offer functionalities such as navigation through hyperlinks, image display, information searches and, through the Common Gateway Interface (CGI), access to information systems such as databases.
- A **configuration/administration client**: Configuration/administration clients are programs which are used to tune server applications. Many objects (services, frameworks) need to have client programs to configure them. For example, in Orbix, a part of the ORB core and the Implementation Repository are implemented by the "`orbixd`" server containing one object. This object is described by the `IT_daemon` IDL interface. But, for this unique object, the Orbix environment provides 14 client tools to configure the daemon (`psit`, `lsit`, `killit`, `catit`, ...). This

high number of client programs is due to the number of services provided by this object type.

Client name	Kind of clients
filemanager, mailtool	navigational specific client
web browser	navigational general purpose client
admintool (SUN)	configuration/administration client

Fig.4: Summary and Examples of Client Types

If the last three kinds of clients are based on the “static” approach, they need to be linked with all object stubs they will use. So, if a new object type or operation are added in the system, the static clients must be redesigned to include the new stubs. This problem does not appear if these client programs are separated from the object types they will invoke. We submit that dynamic invocation and interpreters can be an alternative to writing client programs. We first introduce here a generic script interpreter.

3.2 A Script Interpreter

CorbaScript is an interpreted object-oriented language. Its main functionality is operation invocation onto objects. This interpreter is ORB independent, so a script may be run on different ORBs. It is based on standard CORBA components (see Fig.5):

- the Interface Repository Stubs are used to retrieve IDL informations stored in the IR objects. This permits CorbaScript to control request signatures,
- the ORB Interface is used to manage object references, memory and all CORBA values,
- the Dynamic Invocation Interface is used to build requests regarding script instructions.

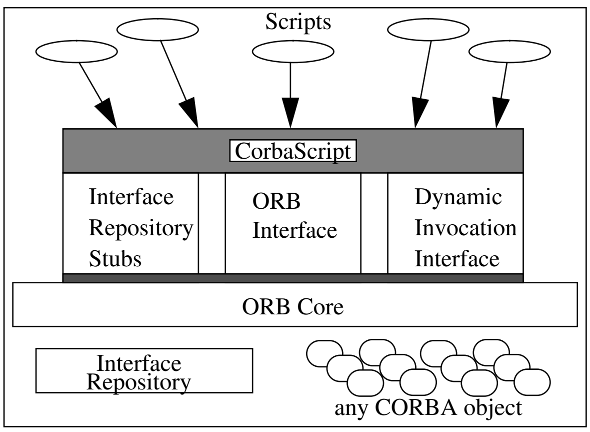

Fig.5: CorbaScript and ORB Architecture

CorbaScript also manages variables and control flow statements. It interprets scripts and constructs requests with DII according to the script informations. The Interface Repository is used to control parameter types and infers types like CORBA structures, arrays, and sequences.

A script is a sequence of instructions. Several kind of instructions are available:

- the *echo* instruction to output any value,
- the assignment instruction (`:=`) to set a value to a variable. The variables are defined at their first use,
- the control flow instructions like the conditional test instruction (*if* `condition` *then* `instructions` *else* `instructions` *end*) and the condititional loop instruction (*while* `condition` *loop* `instructions` *end*),
- the exception management instructions to handle raised CORBA exceptions (*try* `instructions [` *catch* `exception_type instructions]* [`*catchany* `instructions]` *end*), and
- the operation call, the attribute getting or setting instructions onto any CORBA objects (named invocation).

An **invocation** allows users to apply operation on objects. The interpreter dynamically constructs a request according to the user’s input. Then this request is sent to the ORB which executes the request on the right object and returns the result (see later for details).

A script is a set of instructions handling values and variables. A CorbaScript value could be:

- any basic value like integer, float, character or string,
- a set of CorbaScript values like `(10, 'a', "hello")`,
- the result of an operation invocation on an object,
- any CORBA typed value composed of an IDL type and a set of values,
- any expression of values with standard operators (+, or, <=, ...),
- a CORBA object reference.

A CorbaScript **object reference** to a CORBA object is composed of four parts: an IDL object interface, an object name, an application name and a host name; e.g. `bank(mybank:bankSrv:blanche)` refers to an object of type `bank` with the name `mybank`; this object is managed by the `bankSrv` application which runs on the computer blanche. On some ORB, some fields can be empty[1]. References can be stored in shell variables.

As we said before, we have a complete mapping of IDL in CorbaScript so we can handle simple or composed IDL **values** (integer, character, string, sequence, union, ...). The shell performs type inference on the parameters; for example (10, 'hello', True) given as a parameter to a method call is translated into the appropriate IDL structure. The user does not give the IDL types neither for the items in the structure nor for the structure type; all this work is done by the interpreter. In TclDII [Jagannathan 94], the user has to precisely define the IDL type of each item so this tool is hard to use for a non IDL-CORBA expert.

3.3 A CorbaScript Example

The example below shows some invocations of objects to manage bank accounts. First, we bind to the bank object and create two accounts (an `account` and a `depositAccount`). In a second step, we apply operations on accounts and then we display some results.

```
$bank := bank(mybank:bankSrv:blanche)
$acc1 := $bank.newAccount ("Gransart")
$acc2 := $bank.newDepositAccount ("Merle")

$acc1.makeLogdement (1000)
$acc2.makeLogdement (200)

echo $acc1.balance
echo $acc2.describe()
$list := $bank.list_accounts ()
echo $list
```

Fig.6: CorbaScript Example

For example, the invocation `$bank.newDepositAccount ("Merle")` is translated into the following DII calls. The CorbaScript kernel:

- creates an argument list (the `CORBA::NVList` class) for the operation `newDepositAccount` of the `bank` interface,
- sets the first argument of the list (a `CORBA::NamedValue` instance) with the string value `"Merle"`,
- creates a request (the `CORBA::Request` class) onto the object `$bank` and with the previous argument list,
- applies the called operation on the request,
- stores the request result into the variable `$acc2`.

The CorbaScript tool aims at providing a simple language to invoke any operation on objects distributed in a large scale environment. This language hides the complexity needed to use the Dynamic Invocation Interface and Interface Repository.

3.4 A Software Component

In this subsection, we describe some implementation details about CorbaScript. CorbaScript is composed of two main parts: a script analyser and a virtual machine.

The lexical analyser is implemented by the Flex library. This library produces tokens, used by the syntax analyser, from various input streams (e.g. standard input, files, strings). The diversity of input streams allow us to facilitate the reuse of CorbaScript. The syntax analyser is designed with Yacc. The Yacc rules produce a pseudo code.

This pseudo code is interpreted by the CorbaScript virtual machine (VM). The VM instructions get their arguments from a stack. Some thirty instructions compose the VM. Many of them implement arithmetic, logical, comparison operators on stack values. The three main instructions (`op_call`, `op_get`, `op_set`) implement the dynamic invocation mechanism: respectively operation invocation, getting attribute and setting attribute on CORBA objects.

CorbaScript is an interpreter that is flexible and easy to use. CorbaScript is also a C++ component which simplifies the use of DII in future applications. This component is already reused in another tool (CorbaWeb) which allows the user to interact with CORBA objects by the way of a Web browser. This work is presented in the next section.

4 CorbaWeb

The CorbaScript interpreter is a powerful tool to invoke objects, but it needs user program scripts. CorbaWeb aims at providing a generic browser to invoke CORBA objects as easily as users navigate over the World Wide Web (or just Web). It is based on any Web browser (Mosaic, NetScape, HotJava) and a Web server extension through a CGI program. The Web browser manages interactivity with user and sends the user's actions to the Web server. The CorbaWeb program acts as a gateway between the Web and CORBA; it receives the user's actions and translates them into scripts for the CorbaScript component and re-

1. On Orbix all the fields are used; on Orbeline, the hostname field is not used.

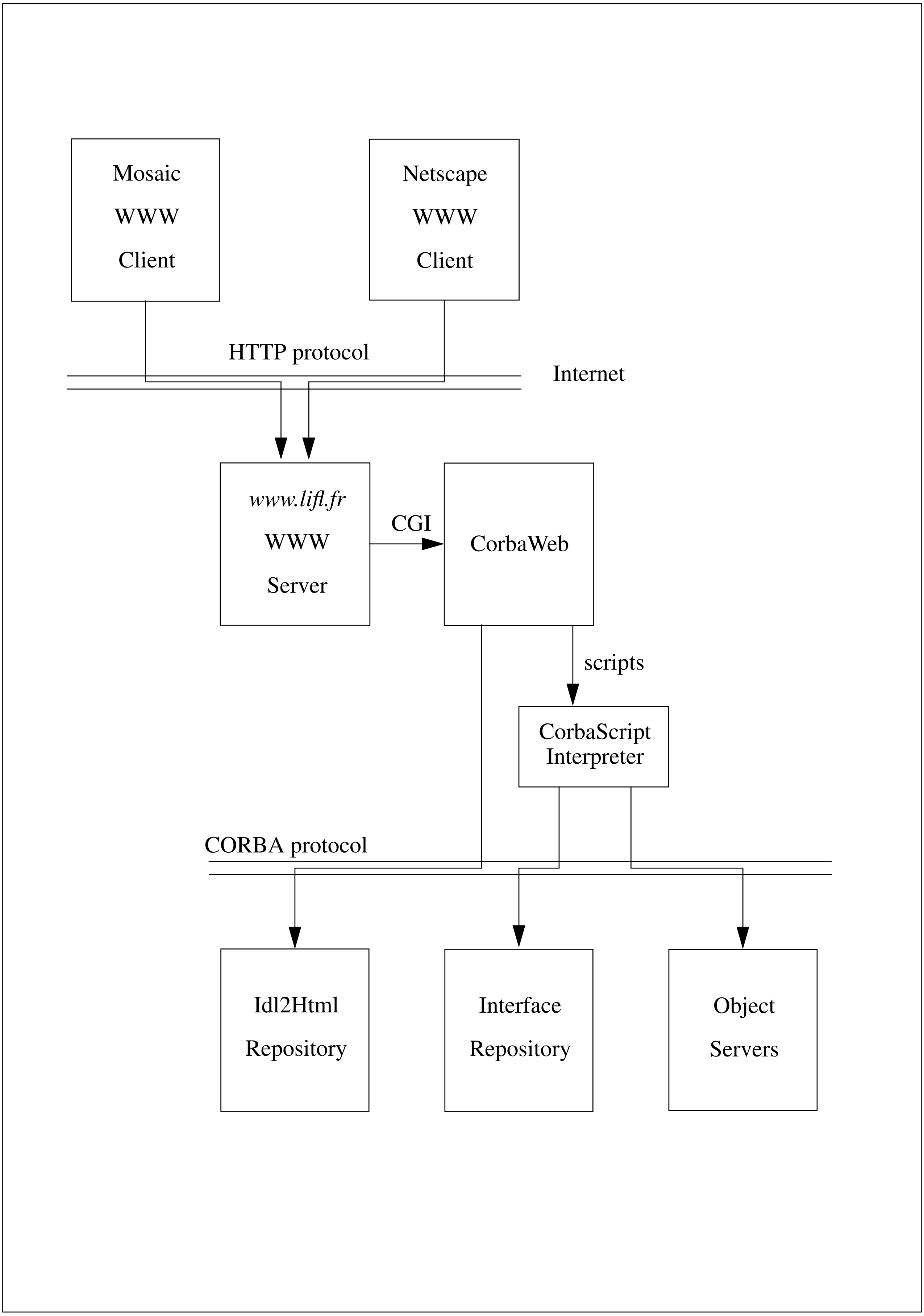

Fig.7: CorbaWeb Architecture

turns a HTML document containing the invocation results.

4.1 The Web Architecture

The World Wide Web is a distributed system, developed at CERN, to connect heterogeneous information databases through a hypertext document metaphor. The Web is a distributed client/server architecture in which clients (often browsers) are allowed to navigate through a graph of multimedia documents (such as texts, images, sounds, videos) stored into WWW servers. The hypertext documents are described in HyperText Markup Language (HTML [Connolly]). HTML permits one to describe document contents and hypertext relations (anchors) between documents.

Clients and servers communicate through the HyperText Transfer Protocol (HTTP [Berns 95]). Clients send HTTP requests and in return, servers reply by sending the requested document. The WWW documents (also resources) are referenced by Uniform Resource Locators (URLs) composed of the server IP address and the resource local name. A requested document can be produced dynamically by a program (a CGI script). Common Gateway Interface (CGI [McCool]) is a standard for external gateway programs to interface with HTTP servers. CGI defines the format of the data stream between the WWW server and gateway programs, and also the environment variables available to gateway programs. The browsers can communicate more information to CGI scripts through HTML Forms (graphical "widgets" displayed into the client window). CorbaWeb is such a CGI program which acts as a gateway between the Web environment and the CORBA environment.

4.2 Why CorbaWeb ?

CorbaScript is a simple shell that requires users to write scripts to invoke objects. CorbaWeb adds navigational functionality through objects. As [Black 94] has pointed out the WWW may be seen as an Object-Oriented Operating System. It furnishes the canonical functionalities of an OOOS:

- a `uniform naming` with the URLs,
- `persistent objects` implemented via http daemons and their host file systems,
- an `invocation meta-protocol` through HTTP and a set of basic operations (GET, POST, ...), and
- `extensibility` via the CGI protocol.

These authors point out some lessons about WWW that can increase the OOOS impacts:

- a `single uniform user interface` to access all objects, (CorbaWeb aims at that), and
- `simple script languages` are powerful ways to incorporate new applications. CorbaScript permits one to access any service implemented by CORBA objects.

So, these are many similarities between WWW and OOOS (like CORBA) and we have tested the mixture of the best of both worlds: CORBA and WWW.

4.3 The CorbaWeb Architecture

The CorbaWeb environment is supported by three components (see Fig.7):

- **the standard WWW technology** where any WWW browser acts as a graphical user interface to browse and to invoke CORBA objects represented both by HTML anchors to access objects and by HTML Forms to invoke operations on them,
- **the CorbaWeb CGI program** which interprets the user's requirements by invoking objects through the CorbaScript component, and generates HTML documents to show request results, and
- **the CORBA environment** which supports the heterogeneous distributed object-oriented system.

CorbaWeb is a C++ program containing the following items:

- the Common Gateway Interface adaptor which allows CorbaWeb to work as a CGI program and to access CGI parameters such as HTML Forms input entries,
- the CorbaWeb component which connects all the other components to allow the CORBA-Web gateway. It translates CGI arguments into a script; this script is executed by a CorbaScript component and a HTML page is generated for the results using the Idl2Html Repository,
- the Idl2Html Repository which maintains information to translate IDL values into their HTML representation, and
- the CorbaScript component which hides the complex DII API, accesses to the Interface Repository and so dynamically furnishes transparent access to all CORBA objects.

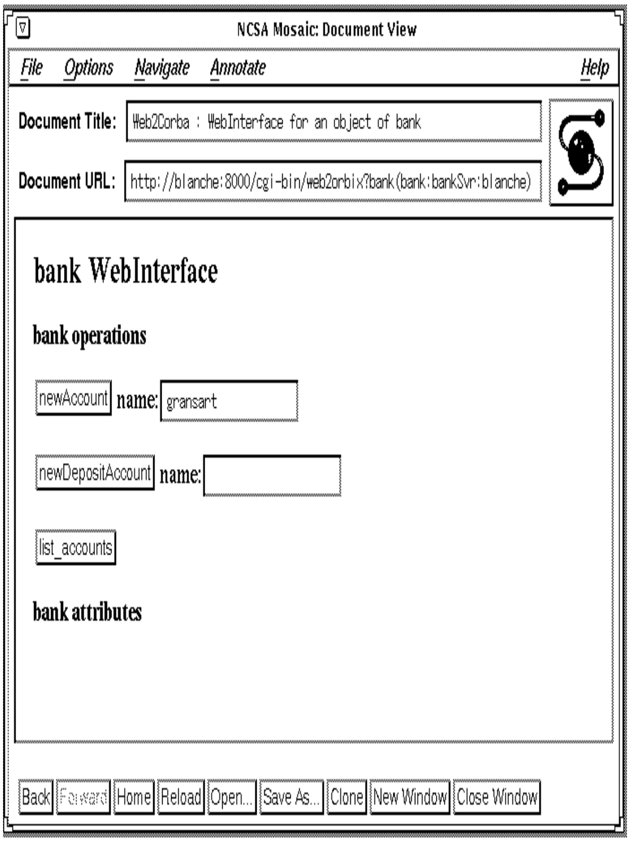

Fig.8.a: HTML Forms of `bank` Interface

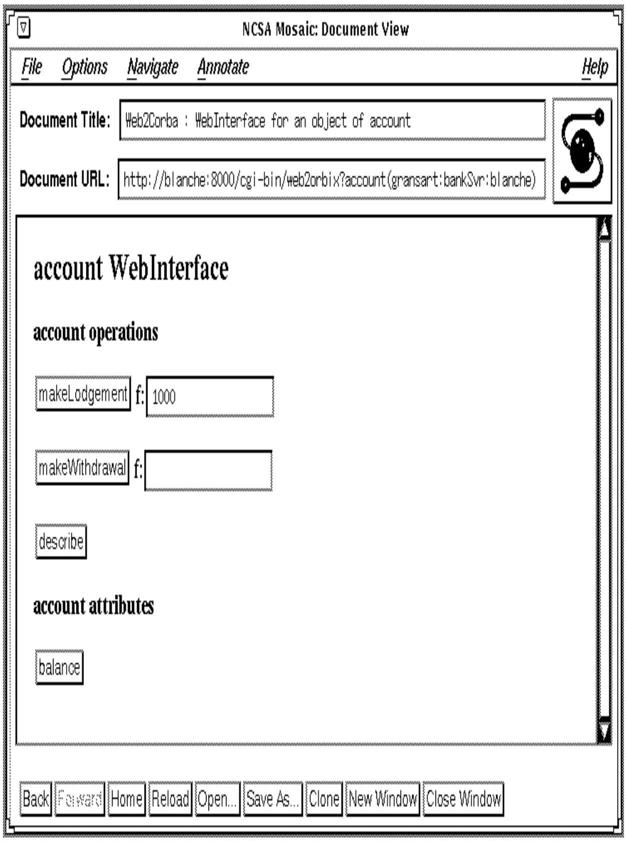

Fig 8.b: HTML Forms of `account` Interface

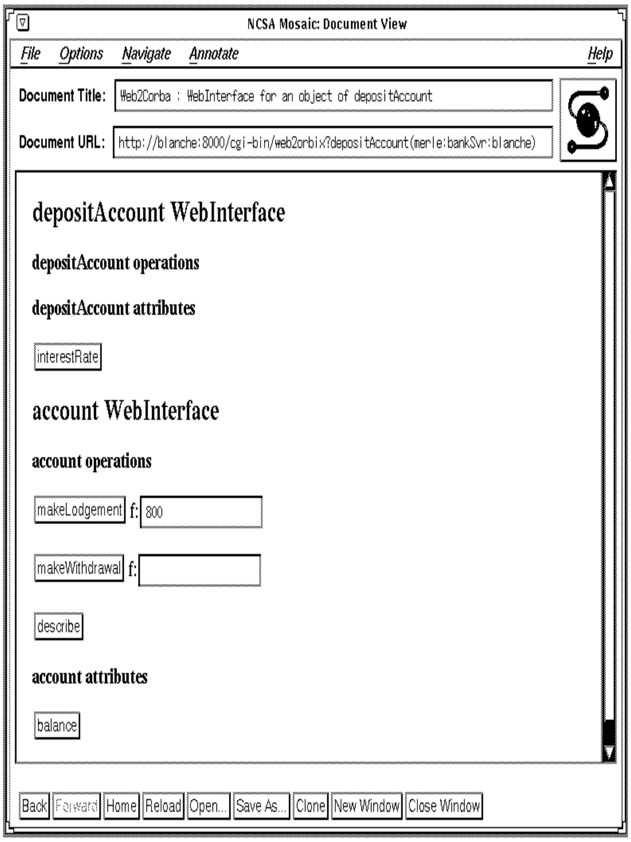

Fig.8.c: A `depositAccount` Object

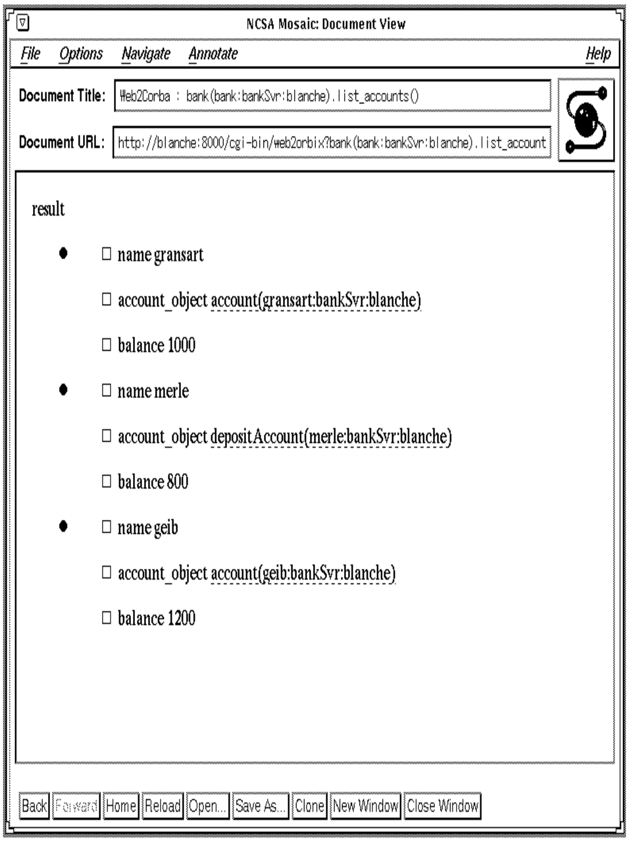

Fig.8.d: Method Invocation Result

Fig.8: CorbaWeb Banking Example

4.4 IDL to HTML Mapping

CorbaWeb allows one to navigate through a CORBA object graph with a Web browser. The Web browser handles the graphical user interface by displaying HTML pages representing object interfaces and invocation results. The mapping between CORBA values such as strings, object references, IDL interfaces and HTML representation is stored into the Idl2Html Repository. When a CorbaWeb component wants to display an IDL typed value, it calls the repository to find what it must display. This consultation is parametrized by the end-user identifier, allowing the generation of the most appropriate representation for the end-user. For instance, an administrator may view all the operations and attributes about an object interface whereas a simple user may not be permitted access to some administration operations. As this repository is a set of CORBA objects, it could be customized using the CorbaWeb environment. This repository contains information to translate all IDL types into HTML:

- the simple types such as integer, string, boolean, enum are represented by HTML text,
- the object references are mapped into HTML anchors. These anchors contain a script allowing access later to the object interface,
- an IDL interface is composed of a set of HTML Forms, one for each operation and attribute. Each form contains a button to execute the script representing the operation (or attribute) and a set of input fields to set the `in` and `in/out` requested parameters. Each input field is represented in accordance to its IDL type, for instance an enum parameter is mapped into an item menu, a structure is mapped into a set of input fields corresponding to each structure field type.
- the arrays, sequences and structures are recursively traversed, each item or field is mapped into HTML (see Fig.8.d).

The anchors are also named URLs, they have the general form `protocol://server/resource`. The first part of a URL defines the protocol to access a resource server (e.g., http, ftp, news). The second part contains the IP address of the server where the resource is stored. And the last part refers to the local resource name on the server. If the resource is a CGI program, the resource name is composed of the program name and the parameters (like `program?parameters`). Figure 8 shows some URL examples generated by CorbaWeb. The anchors generated by CorbaWeb reference the CGI program[1] and the script part is interpreted by the CorbaScript component.

4.5 The Banking Example

In this section, we show how the CorbaWeb environment works with the banking IDL definitions described in section 2.3. The snapshots (Fig.8) show how a user navigates and invokes objects through the NCSA Mosaic browser. The CorbaWeb environment automatically generates these HTML pages.

In this example, CorbaWeb is installed into the Web server located on the host blanche, on IP port 8000 and the CorbaWeb program is named web2orbix, so each URL in the following sentences are implicitly prefixed by "http://blanche:8000/cgi-bin/web2orbix?".

In Fig.8.a, the browser requests the URL "bank(bank:bankSvr:blanche)" and then CorbaWeb sends back an HTML page containing the set of HTML Forms to invoke "bank" interface operations onto the object bank:bankSvr:blanche (the first name is the object local name and the second is the server name; a third argument could be given to specify the host).

Each operation is represented by a specific HTML Form with input fields. Read-only attributes (see Fig.8.b) are shown by a button to allow access to the attribute value. If the user selects a Form button then the browser sends to the server a request containing an URL and input field values. The server transmits this information to the CGI CorbaWeb program. CorbaWeb constructs a script which is executed by a CorbaScript component. In return, CorbaWeb generates an HTML page containing the results.

Information in Fig.8.c is obtained because the user has invoked the button newDepositAccount. The bank object creates a new `depositAccount` object and returns it by reference. Then, CorbaWeb generates an HTML page to represent this object reference by consulting the Idl2Html repository. Note that this page contains the `depositAccount` interface features as well as the `account` features.

Fig.8.d illustrates the recursive path of structured IDL types; the `list_accounts` operation invocation returns a sequence of `description` structures. This structure is composed of an object reference `account_object` (see the HTML anchor for object references) and of other information fields.

As shown in this example, CorbaWeb is a generic object browser through the Web environment. Presently, it can browse many CORBA frameworks: for instance, the naming service is an object graph which

1. cgi-bin/web2orbix for the Orbix implementation.

can now be updated through Mosaic using CorbaWeb. In most ORB implementations, the Interface Repository has no browser to navigate into. One can use CorbaWeb for this purpose.

5 Future Works

GOODE is implemented in the C++ language on Orbix and Orbeline ORBs. This leads to two different implementations of the CorbaScript kernel, but CORBA 2.0 specification will have to resolve this problem. This will increase the portability of GOODE onto other ORB implementations.

CorbaWeb must be customized with the Idl2Html repository. This customization allows us to represent information more specifically for each framework and for each end-user. At present, the Idl2Html Repository is not totally implemented we must integrate the end-user issue. We will develop it in a real application context to discover all parameters needed.

CorbaWeb is a powerful tool to browse CORBA objects and it can become an interesting tool to structure information systems managed by WWW servers [Edwards 95]. Our plan is to develop what we call WebCorba (use of CorbaWeb for the Web infrastructure) which will manage semi dynamic HTML documents: documents mixing HTML contents and CorbaScript code. Then WebCorba will allow the integration of dynamic information with static Web services implemented by CORBA objects.

We will have to study another major problem, namely, a solution to end-to-end access control: Objects must be accessible only by authorized people and processes. At present, thanks to CorbaWeb, any object can be accessed by anybody. Global access control requires a mixture of different access control policies such as Web authentication, CORBA security services and local object control access.

GOODE is a new approach to support generic applications (CorbaScript, CorbaWeb). GOODE is experimented above CORBA but it could be supported by other object-oriented systems. These systems must offer two components: an object type repository and a dynamic invocation mechanism. Object Linking and Embedding in Microsoft Windows [Brockschmidt 94] could be a good candidate because it integrates OLE Automation (an invocation mechanism) and the Type Library (an object type repository).

In prospect, we plan to develop a cooperative generic browser with the GOODE approach. It will act as a cooperative object browser for the OSACA environment (objects, threads [Geib 95] and groups [Namyst 94]).

6 Implementations and Conclusion

GOODE has been implemented and is operational on a SUN Sparc station under Orbix 1.3 for Solaris 2.4. CorbaScript and CorbaWeb prototypes are available. Prototype implementations consist of 8,000 lines of C++ code. Information in Figures 8 is generated by the current CorbaWeb prototype.

We have presented a new approach to the use of complex object systems like CORBA: the Generic Object-Oriented Dynamic Environment (GOODE) explores a new application domain based on the dynamic invocation mechanism of CORBA. We have introduced in this paper, first, CorbaScript, which allows one to simplify dynamic invocation use and, second, CorbaWeb which allows one to browse dynamically any objects through Web browsers. So, GOODE integrates the Web world and the distributed object world seamlessly.

At present, GOODE is a collection of generic tools. It will become a generic toolbox reusable in other contexts. These projects should help to define a generic toolbox with attractive components to build generic applications over large scale distributed systems.

7 References

[Berns 95] T. Berners-Lee, R.T. Fielding, H. Frystyk Nielsen, *"HyperText Transfer Protocol - HTTP/1.0"*, Internet-Draft work in progress, Internet Engineering Task Force, <URL:ftp://nic.nordu.net/internet-drafts/draft-ietf-http-v10-spec-00.ps>.

[Black 94] A.P. Black, J. Walpole, *"Objects to the rescue! or httpd: the next generation operating system"*, ACM-SIGOPS 94, Sixth SIGOPS European Workshop, Wadern Germany, September 1994.

[Brockschmidt 94] K. Brockschmidt, *"Inside OLE 2"*, Microsoft Press, Redmond, 1994.

[Connolly] D. Connolly, *"HyperText Markup Language (HTML)"*, <URL:http://www.w3.org/hypertext/WWW/MarkUp/MarkUp.html>.

[Edwards 95] N. Edwards, *"Object Wrapping (for WWW) - The Key to Integrated Services?"*, APM.1464, APM Ltd., Cambridge U.K. April, 1995.

[Fresco 94] X Consortium Working Group, *"Fresco Sample Implementation Reference Manual"*, Draft, Version 0.7, April 1994.

[Geib 95] J.M. Geib, C. Gransart, C. Grenot, P. Merle, *"(Object and Thread)-Oriented Distributed Programming"*, Object Based Parallel and Distributed Computing, Workshop, Tokyo, Japan, June 21-23, 1995.

[Jagannathan 94] G. Alamsi, A. Suvaiala, C. Goina, C. Cascaval, V. "Juggy" Jagannathan, *"TclDii: A TCL Interface to the Orbix Dynamic Invocation Interface"*, <URL:http://www.cerc.wvu.edu/dice//iss/TclDii/TclDii-Doc.ps>, TR, 1994.

[Kessler 94] P. Kessler, *"A Client-Side Stub Interpreter"*, Proceeding of ACM Workshop on Interface Definition Language, January 1994.

[Manola 93] J.R. Nicol, C.T. Wilkes, F.A. Manola, *"Object Orientation in Heterogeneous Distributed Computing System"*, IEEE Computer 26, June 1993.

[McCool] R. McCool, *"The Common Gateway Interface"*, <URL:http://hoohoo.ncsa.uiuc.edu/cgi/overview.html>.

[Meyer 88] B. Meyer, *"Object-Oriented Software Construction"*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Namyst 94] P. Merle, R. Namyst, J.M. Geib and J.F. Méhaut, *"Objects Spaces, Cooperation Spaces and Groups"*, ACM-SIGOPS 94, Sixth SIGOPS European Workshop, Wadern Germany, September 1994.

[OMG 92] Object Management Group, R.M. Soley, *"Object Management Architecture Guide"*, OMG Document 92.11.1 (Revision 2.0), OMG, September 1, 1992.

[OMG 93] Object Management Group and X/Open, *"The Common Object Request Broker: Architecture and Specification"*, OMG Document 93.12.43 (Revision 1.2), OMG, Framingham, Mass., 1993.

[ORBeline] *"ORBeline User's Guide"*, PostModern Computing, 1994.

[Orbix] *"ORBIX^{TM} Programmer's Guide and Advanced Programmer's Guide"*, Release 1.3, IONA Technologies Ltd, April 1995.

[Wegner 90] P. Wegner, *"Concepts and Paradigms of Object-Oriented Programming"*, ACM OOPS Mesenger, Vol 1 Number 1, Aug 1990.