

PROJECT REPORT ON

COBMAF – A CORBA Based Multi-Agent Framework Version 2.0

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF ENGINEERING
IN
INFORMATION TECHNOLOGY
2005-2006

SUBMITTED BY:

**ASHAY KHARPATE, KARAN KAMDAR,
RAHUL IYER & SUJEET GHANVAT**

UNDER THE GUIDANCE OF:

Prof. Mrs. Nupur Giri
Department of Computer Engineering
V.E.S. Institute of Technology
University of Mumbai

Vivekanand Education Society's

INSTITUTE OF TECHNOLOGY

Sindhi Society, Chembur, Mumbai-400 071



CERTIFICATE OF APPROVAL OF PROJECT WORK

This is to Certify that
Mr/Miss _____ has satisfactorily
carried out the project work entitled
_____ in partial
fulfillment of the B.E. Degree in _____ of the
University of Mumbai, Maharashtra state during 2005-
2006

PRINCIPAL

HEAD OF DEPARTMENT

PROJECT GUIDE

EXAMINER

ACKNOWLEDGEMENTS

The COBMAF Version 2.0 team is greatly thankful to its internal project guide Prof. Nupur Giri who so ably supported us throughout the year. She taught us the value of one's work and gave us the motivation to try harder every single time, with the accomplishments of this project truly representing her teachings. It is under her unwavering guidance and blessings that we could achieve most of the objectives that we set out for this BE project.

The COBMAF Version 2.0 team also would like to thank its senior batch of student researchers namely Devraj Bhramachari & Manas Khadilar who laid the foundation for this BE project in the form of Virtual Enterprise over COBMAF. Their documentation and source code proved to be so worthwhile that we couldn't have realized any of our objectives without their effort. We also thank you guys for providing us with electronic and telephonic support during the harder times that we went through.

TABLE OF CONTENTS

1. COBMAF Version 1 Revisited.....	1
1.1 Genesis.....	1
1.2 Abstract.....	2
1.3 Project Requirements.....	4
1.3.1 The CORBA Framework.....	4
1.3.2 The COBMAF Framework.....	14
2. COBMAF Version 2 – The Essential Upgrade.....	18
2.1 Overview.....	18
2.2 Problem Definition.....	19
3. Porting from BOA to POA.....	20
3.1 What is the Portable Object Adapter (POA)?.....	20
3.2 How the ORB uses POA?.....	20
3.3 Why the need to port?.....	20
3.4 Guidelines for migration from BOA to POA	21
3.5 How the porting was performed?.....	22
4. Directory Facilitator.....	24
4.1 Overview.....	24
4.2 Architecture.....	25
4.3 Management Functions Supported by the Directory Facilitator.....	25
4.4 DF Object Descriptions.....	26
4.4.1 Directory Facilitator Agent Description.....	27
4.4.2 Service Description.....	28
4.5 DF Function Descriptions.....	29
4.5.1 Registration of an Object with the DF.....	30
4.5.2 Deregistration of an Object with the DF.....	30
4.5.3 Modification of an Object Registration with the DF.....	31
4.5.4 Search for an Object Registration with the DF.....	31
4.6 Implementing the DF.....	32
4.7 Agent-DF & Agent-DF-DF-Agent Interaction Diagrams.....	35
5. Ontology.....	38
5.1 Overview.....	38
5.2 Introduction.....	39
5.3 Rationale.....	39
5.4 COBONTO - The Ontological Framework for COBMAF.....	41

6. The COBMAF-Swing Graphical User Interface (CSGUI).....	52
6.1 What is the CSGUI?.....	52
6.2 Why the need to upgrade?.....	52
6.3 AWT versus Swing.....	52
6.4 Why Swing was chosen?.....	53
7. The COBMAF package structure (CPS).....	55
7.1 Overview.....	55
7.2 cobmaf.content.onto package.....	55
7.3 cobmaf.core package.....	56
7.4 cobmaf.core.behaviours package.....	57
7.5 cobmaf.gui package.....	58
7.6 cobmaf.lang.acl package.....	59
7.7 cobmaf.lang.csl package.....	59
7.8 cobmaf.lang.kql package.....	60
7.9 myInterface package.....	60
8. The COBMAF WEB MODEL (CWM).....	62
8.1 Overview.....	62
8.2 Architecture.....	63
8.3 The DF Server side Architecture.....	64
8.4 The Local AMS (Client-Side Architecture).....	65
8.5 How our Web Model works?.....	66
9. Concluding Features and Test Parameters.....	68
9.1 Transparency Issues.....	68
9.2 Scalability.....	70
9.3 Failsafe/Self-configuration Nature.....	72
9.4 Negotiation Strategy (DF-DF).....	73
9.5 Co-ordination/Synchronization Model.....	75
9.6 Naming Service/Discovery Support.....	76
9.7 Interoperability & Portability.....	78
9.7.1 Interoperability.....	78
9.7.2 Portability.....	79
10. Future Scope.....	81
11. Screenshots.....	82

COBMAF: A CORBA BASED MULTI AGENT FRAMEWORK VERSION 2.0

Nupur Giri¹, Karan Kamdar.¹, Ashay Kharpatе¹, Sujeet Ghanvat¹, Rahul I.¹

¹ Computer Engineering Department, Vivekanand Education Society's Institute of Technology, Chembur, Mumbai, India.

nupur.giri@gmail.com, karan@stople.com, ashaykh@gmail.com, eghanvat@yahoo.com, dude.rahul@gmail.com

1. COBMAF VERSION 1 REVISITED

1.1 Genesis

The success of any business organization today lies not only in its strong foundation and individual genius but also in its ability to collaborate with the remaining business world and get the best out of it. With the arrival of networks and distributed systems, geographical dispersion of business organizations poses little hindrance now. Today, Virtual Enterprises (VEs) offer a platform for numerous inter-dependent organizations to come together, share resources and collaborate.

A Multi Agent System (MAS) represents a virtual society of pro-active participants reflecting situations in real-life VEs. The extensive research on Multi Agent Systems (MAS) going on in our Institute for the last few years offered us a big initiative to use this technology in implementing a VE.

Even though Multi Agent Systems accommodate heterogeneity in terms of function, there is still a huge restriction on the heterogeneity of the design of the individual agents themselves i.e. they need to be homogenous in terms of language and platform. After working on a Java-RMI based client-server system last year we noticed several short-comings of the technology and realized the importance of having a language and platform independent architecture for heterogeneous servers and clients to communicate. Such architecture is only offered by CORBA.

One major incentive that sealed the decision of this project was given by our external guide, who has personally worked in the same arena and created a Virtual Enterprise using a custom-made language called VEML which incorporated features of KQML.

With ideas of improving the current implementations of VEs, strongly supported with the knowledge gained from our previous work, we decided to undertake this project. We have built a Virtual Automobile Manufacturing Enterprise using a CORBA based Multi Agent System.

1.2 Abstract

The rapid advancements in networking technologies have created the opportunity to use the combined computing and communication resources of computers and devices of different enterprises. These different enterprises, with their unique specializations and competency skills, can agglomerate to create a virtual environment that caters to the achievement of an integrated larger goal.

A **Virtual Enterprise (VE)** is an enterprise that has no resource of its own but consists of shared and coordinated activities that utilize the resources of the participating enterprises. These enterprises are distributed in nature. Adding together the resources of all constituent enterprises requires technology for networking, communication, co-ordination and decision-making.

The interaction between the enterprises in a VE is achieved through the use of software agents. In a multi-agent environment, each participating enterprise houses one or more agents. The agents perform certain specialized functions, both in terms of computation and communication, on behalf of their host enterprises.

A variety of agent development environments exist that facilitate agent development in the one language. However, VE being a heterogeneous environment, needs to be provided with an Application Programming Interface(API) to enable the ease of agent design and efficient communication between agents coded in different languages on different platforms. To allow the heterogeneity the API uses the **CORBA** (Common Object Request Broker Architecture) middleware to build VE.

An agent can be considered as an intelligent object that can live anywhere on a network. By using CORBA, the language and compiler used to create the agents will

be a transparent entity during inter-agent communication. In this type of interaction, an agent, irrespective of the language it has been coded in, need not know about the language or the platform of the agent it is communicating with.

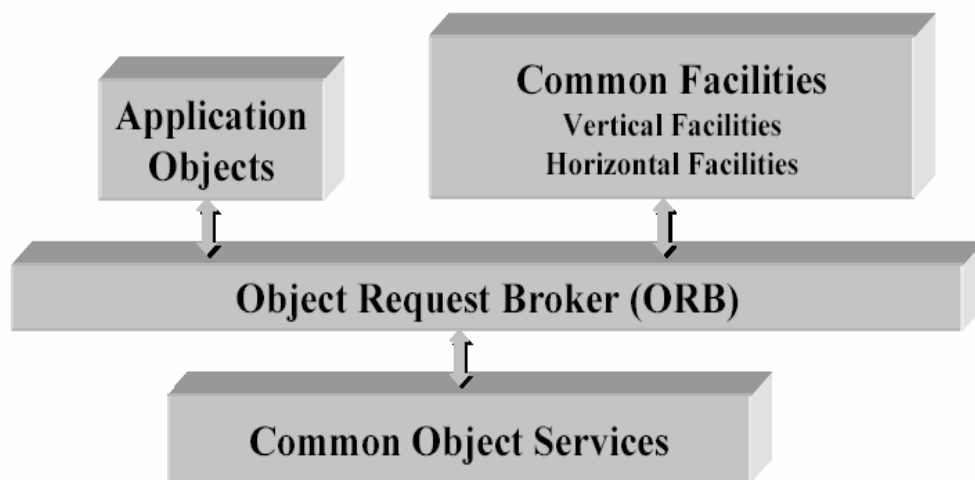
In order to illustrate the working of such a communication model, our project will make use of an **automobile manufacturing** application of a VE System. An automobile manufacturing application requires different types of agents for different types of duties like –

1. Sales Agent.
2. Procurement Agent.
3. Scheduling Agent.
4. Resource Management Agent.

1.3 Project Requirements

1.3.1 The CORBA Framework

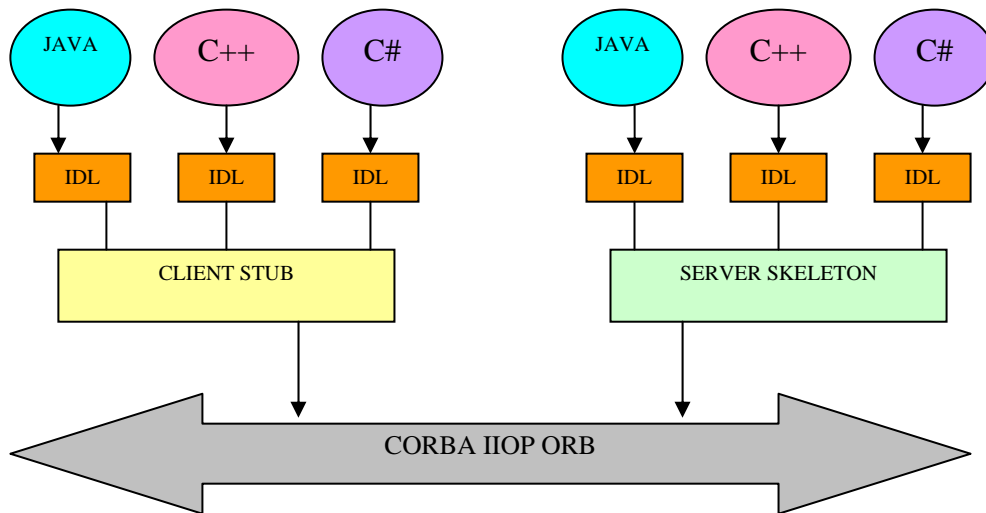
To incorporate more flexibility into the implementation of agents in the Virtual Enterprise the CORBA (Common Object Request Broker Architecture) platform is used as the middleware. This allows communication between Agents implemented in any language, on any platform and running on any operating system. Such **heterogeneous agents** provide autonomy, simplicity of communication, computation and well-developed semantics. They ensure the necessary flexibility and dynamism of a VE System.



The common Object Request Broker Architecture (CORBA) is a product of a consortium – called the Object Management Group (OMG) – that includes over 800 companies, representing the entire spectrum of the computer industry. The CORBA object bus defines the shape of the components that live within it and how they interoperate. Consequently, by choosing an open object bus, the industry is also choosing to create an open playing field for components. What makes CORBA so important is that it defines middleware that has potential of subsuming every other form of existing client server middleware. CORBA uses objects as a unifying metaphor for bringing existing applications to the bus. The specification is always separated from implementation. This lets you incorporate existing systems within the bus.

CORBA creates interface specifications, not code. The interfaces it specifies are always derived from demonstrated technologies submitted by member companies specifications are written in a neural interface definition language (idl) which

defines the components boundaries that is, its contractual interfaces with potential across languages, tools, operating systems and networks.



CORBA objects are blobs of intelligence that can live anywhere on a network. They are packaged as binary components that remote clients can access via remote method invocations. Both the language and the compiler can be used to create server objects are totally transparent to clients. Clients do not need to know where the distributed objects reside on or which operating system it executes on. It can be on the same process or on a machine that sits across on an intergalactic network. In addition client does not need to know how the server object is implemented. Thus the interface serves as a binding contract between the clients and server.

LANGUAGE INDEPENDENCE BY CORBA

CORBA ORB ARCHITECTURE

The following figure illustrates the primary components in the CORBA ORB architecture. Descriptions of these components are available below the figure.

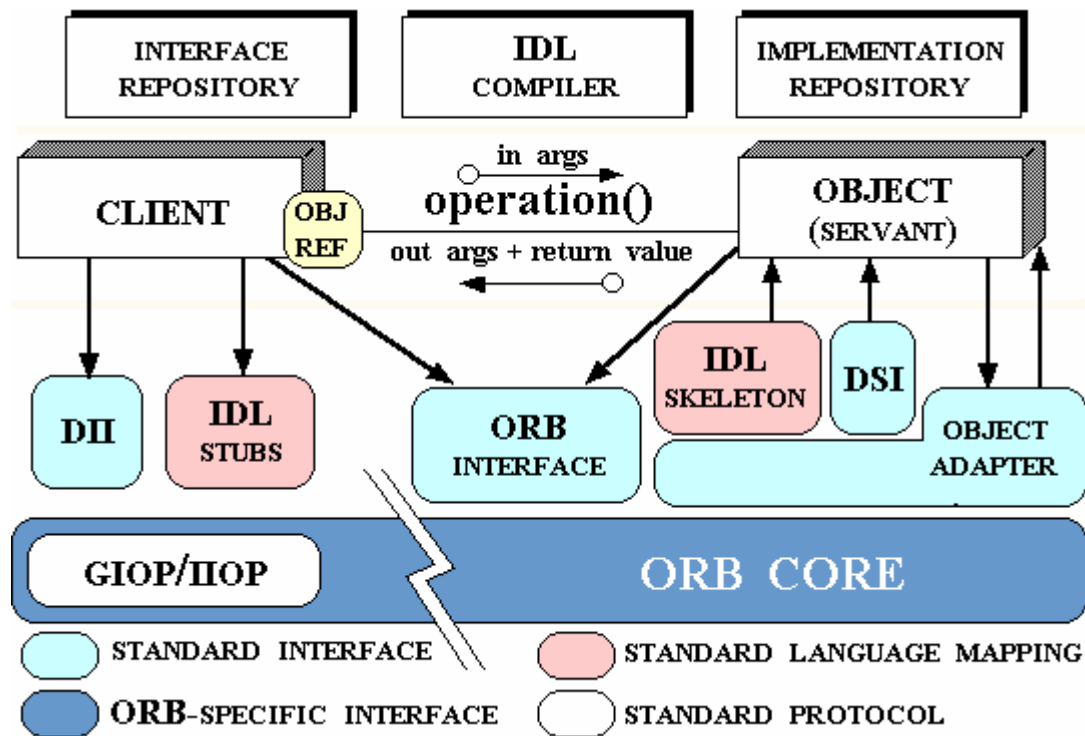


Figure 2. CORBA ORB Architecture

- **Object:** This is a CORBA programming entity that consists of an identity, an interface, and an implementation, which is known as a Servant.
- **Servant:** This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.
- **Client:** This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., `obj->op(args)`. The remaining components in Figure 2 help to support this level of transparency.
- **Object Request Broker (ORB):** The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the

object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

- **ORB Interface:** An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

- **CORBA IDL Stubs and Skeletons:** CORBA IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

- **Dynamic Invocation Interface (DII):** This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking deferred synchronous (separate send and receive operations) and one-way (send-only) calls.

- **Dynamic Skeleton Interface (DSI):** This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

- **Object Adapter:** This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object

implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

These Object Adapters are of two types:--

1. The Basic Object Adapter (BOA)
2. The Portable Adapter (POA)

THE BASIC OBJECT ADAPTER (BOA)

The Basic Object Adapter, or BOA, provides several important functions to client applications and the object implementations they use, including:

- ❖ Providing several policies for activating object implementations, and determining how client applications can access these implementations.
- ❖ Registering object implementations with OS Agent.
- ❖ Installing and registering the object with the Implementation Repository, and activating the object upon client request, with the Object Activation Daemon.
- ❖ Storing information about object implementations residing on a server with the Implementation Repository.

THE PORTABLE OBJECT ADAPTER (POA)

According to the specification, "The intent of the POA, as its name suggests, is to provide an object adapter that can be used with multiple ORB implementations with a minimum of rewriting needed to deal with different vendors' implementations." However, most CORBA products do not yet support the POA.

The POA is also intended to allow persistent objects -- at least, from the client's perspective. That is, as far as the client is concerned, these objects are always alive, and maintain data values stored in them, even though physically, the server may have been restarted many times, or the implementation may be provided by many different object implementations.

The POA allows the object implementer a lot more control. Previously, the implementation of the object was responsible only for the code that is executed in response to method requests. Now, additionally, the implementer has more control over the object's identity, state, storage, and lifecycle.

The POA has support for many other features, including the following:

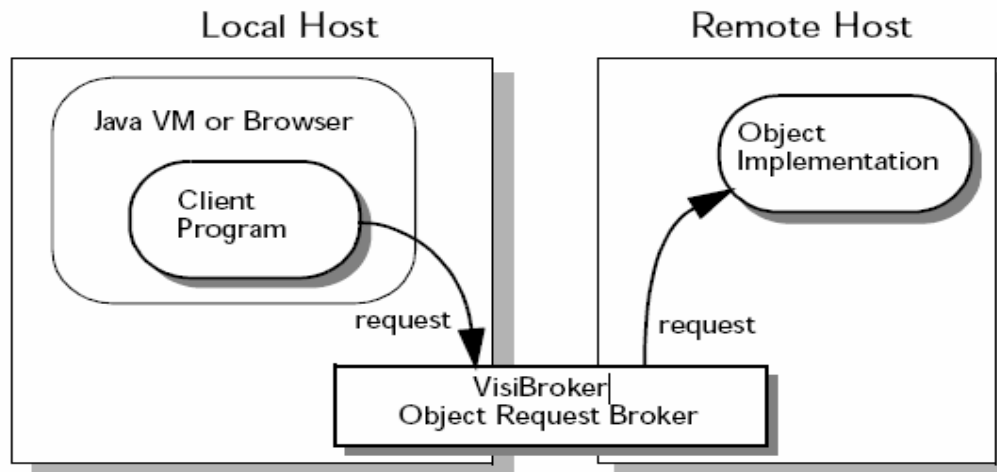
- ❖ Transparent object activation.
- ❖ Multiple simultaneous object identities.
- ❖ Transient objects.
- ❖ Object ID namespaces.
- ❖ Policies including multithreading, security, and object management.
- ❖ Multiple distinct POA in a single server with different policies and namespaces.

BORLAND'S VISIBROKER SOFTWARE

VisiBroker is a complete CORBA Object Request Broker (ORB) and supports a development environment for building, deploying, and managing distributed object applications that are interoperable across platforms. Objects built with VisiBroker are easily accessed by Web-based applications that communicate using the OMG's Internet Inter-ORB Protocol (IIOP), the standard for communication between and among distributed objects running on the Internet, intranets, and in enterprise computing environments. VisiBroker has a native implementation of IIOP, ensuring high-performance, interoperable distributed applications.

The VisiBroker Object Request Broker (ORB) connects a client program (which may be a Java applet or application), running on a Java virtual machine or in a Java-enabled browser, with the objects it wishes to use. The client program does not need to know whether the object resides on the same computer or is located on a remote

computer somewhere on the network. The client program only needs to know the object's name or the object reference for the object and understand how to use the object's interface. The ORB takes care of the details of locating the object, routing the request, and returning the result. The ORB is not a separate process. It is a collection of Java objects and network resources that integrates within end-user applications, and allows your client applications to locate and use objects.



VISIBROKER FEATURES

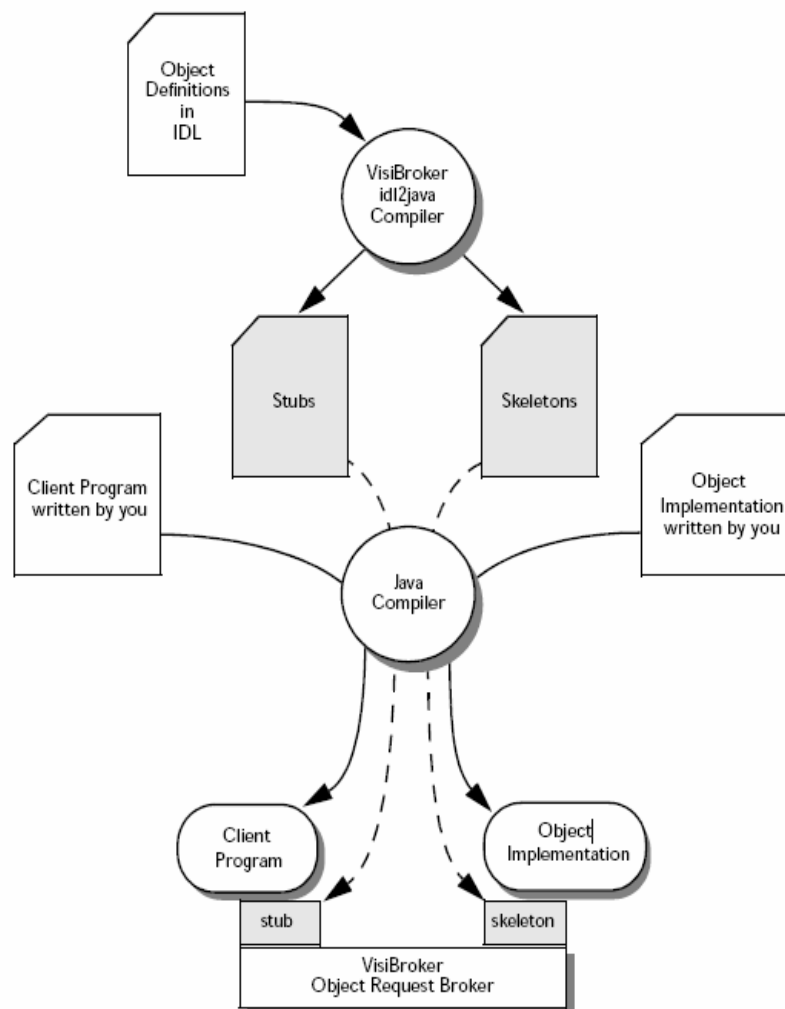
VisiBroker has several key features.

- ❖ Interface Repository
- ❖ Dynamic Invocation Interface
- ❖ Dynamic Skeleton Interface
- ❖ Smart Binding
- ❖ Smart Agents
- ❖ Object Activation Daemon (OAD)
- ❖ Enhanced Thread and Connection Management
- ❖ Location Service
- ❖ Enhanced idl2java Compiler
- ❖ Object Request Debugger
- ❖ Java Ease-of-Use
- ❖ Smart Stubs
- ❖ Interceptors
- ❖ Communication Event Handlers
- ❖ Gatekeeper (optional feature)

DEVELOPING APPLICATIONS WITH VISIBROKER

The first step to creating an application with VisiBroker is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). The IDL can be mapped to a variety of programming languages.

The interface specification you create is used by the VisiBroker idl2java or idl2cpp compiler to generate stub routines for the client program and skeleton code for the object implementation. The stub routines are used by the client program for method invocations. You use the skeleton code, along with code you write, to create the server that implements the object. The code for the client and object, once completed, is used as input to your Java or CPP compiler to produce a Java or CPP application and an object server. For an illustration of these general steps, see Figure below.



STEPS TO BUILD AN APPLICATION

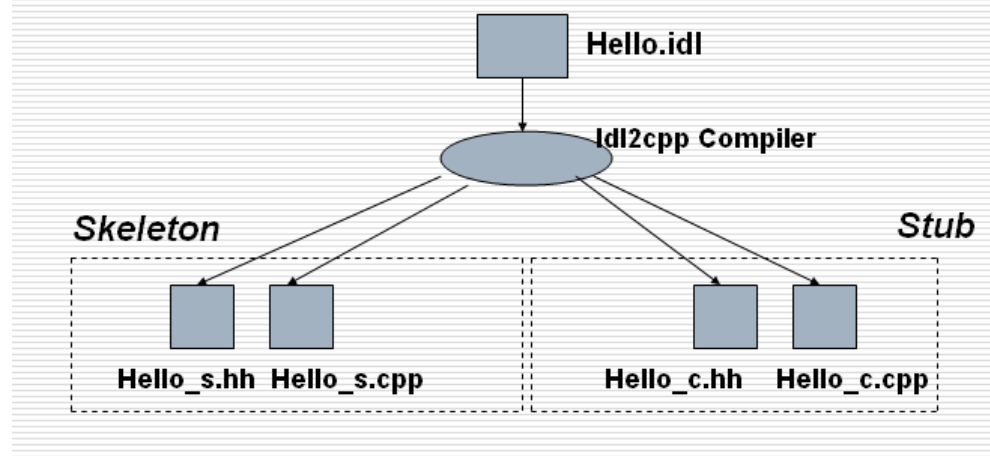
1. Write the Interface using the IDL
2. Implement servant methods.
3. IDL2language compilation.
4. Write the server code.
5. Write client code with method invocation request.
6. Compile the codes
7. Run the Server and the Client.

UNDERSTANDING THE IDL2JAVA AND IDL2CPP COMPILERS BY VISIBROKER

Example: The *Hello.idl* File

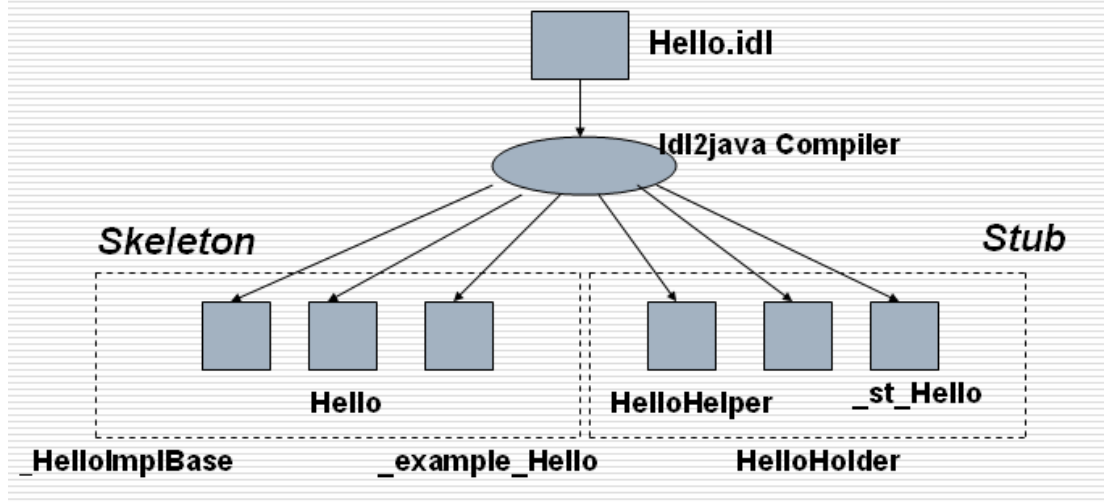
```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

Visibroker: IDL2CPP compiler



The idl2cpp Compiler produces the stub and skeleton files as shown.

Visibroker: IDL2JAVA compiler



The idl2java Compiler produces the stub and skeleton files as shown.

CODE SNIPPET (CLASS *agent.java*):

```
org.omg.CORBA.ORB orbSales = org.omg.CORBA.ORB.init((String[])null,null);
org.omg.CORBA.BOA boaSales= orbSales.BOA_init();
myImpl impl= new myImpl(agentName);
boaSales.obj_is_ready(impl);
boaSales.impl_is_ready();
```

- a: Initializing the ORB.
- b: Initializing the BOA.
- c: Creating an object of class myImpl.
- d: exporting the object to the ORB.
- e: Waiting to service requests.

1.3.2 The COBMAF Framework

1.3.2.1 Application Layer

The application layer can be any virtual enterprise system requiring heterogeneity, distribution, authentication and privacy. From a business perspective it involves contracts, cross-organizational management and statutory obligations. Such an enterprise, implemented as a collection of multi agents, build using the COBMAF framework, allows heterogeneity as well as agents communication using message passing according to FIPA ACL. The paper explains “VAME: Virtual Automobile Manufacturing Enterprise” a globally distributed system. The agents invoked using COBMAF– API use domain specific performatives for interaction.

1.3.2.2 COBMAF : Framework Layer

This is heart of the entire system. The framework presents itself in the form of a set of APIs that the application developers can utilize to create agents. The framework is implemented in various languages so that agents can be language independent and yet share an identical structure and capabilities. The layer provides a mapping between the higher level agents that use message passing through FIPA ACL performatives and the lower level messaging layer that uses middleware functionalities through method invocation primitives.

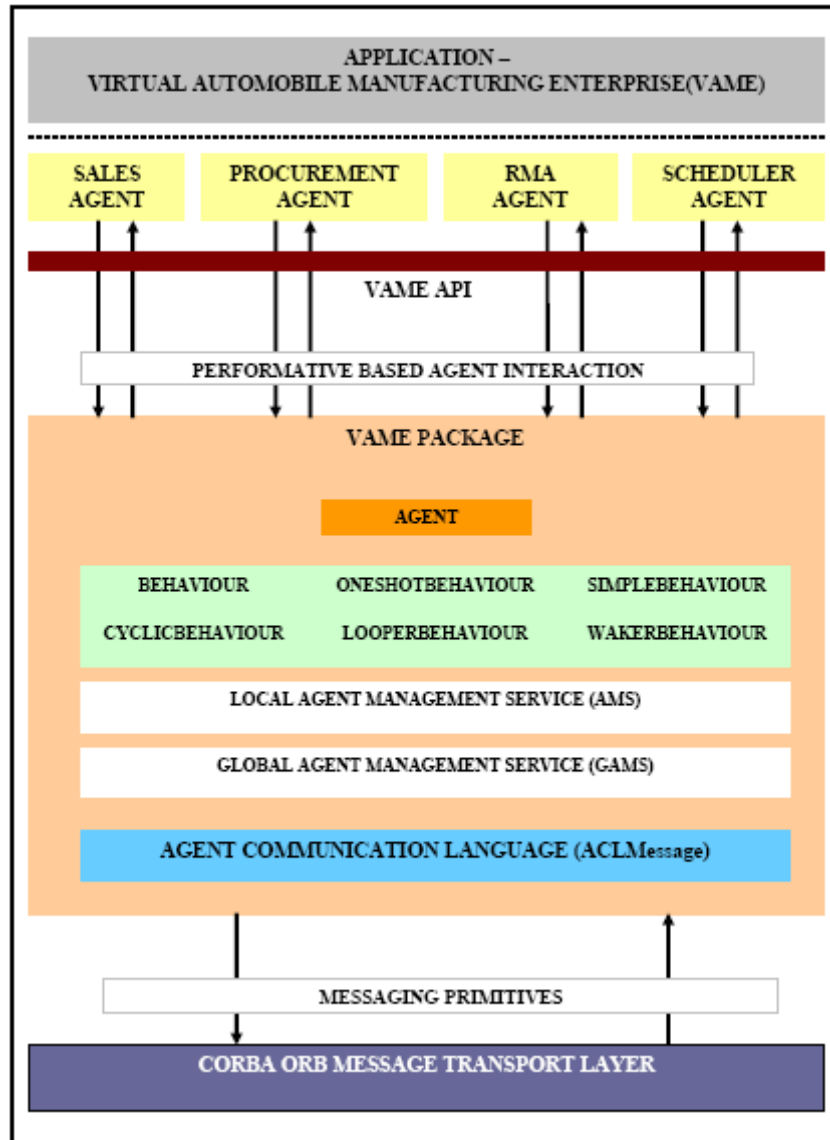


Fig.1.3.2.1: Distributed System architecture for a Virtual Enterprise

1.3.2.3 CORBA message transport layer

The lowest layer is a message transport layer that handles the transmission and reception of messages from the higher layers, translating higher level agent messages into lower level method invocation primitives across a heterogeneous medium. This layer is opaque to the existence of agents in the higher layers. It concerns with the task of delivering a message object wrapped as a CORBA object that makes use of the ORB to invoke the message passing primitives defined using the CORBA IDL. These message passing primitives provided by the underlying message transport layer are:

```
send(Message);
```

```
recv();
```

Here, the '*send(Message)*' primitive takes an object of the Message class as input parameter and sends it to the destination through the CORBA ORB. The '*recv()*' primitive receives a message from the destination. By using wrapper class over these basic message passing primitives the application developer has full access to the basic services of the underlying middleware as well as additional services such as:

- Implicit initialization of the sender address of the message initiating agent.
- Setting the content of the message.
- Setting the Performative for the message.
- Adding one or more receivers for the message.
- Unicasting or Multicasting messages.
- Retrieving the content of the message.
- Retrieving the sender identity of the message.
- Retrieving the performative of the message.

1.3.2.4 The Framework Layer: COBMAF

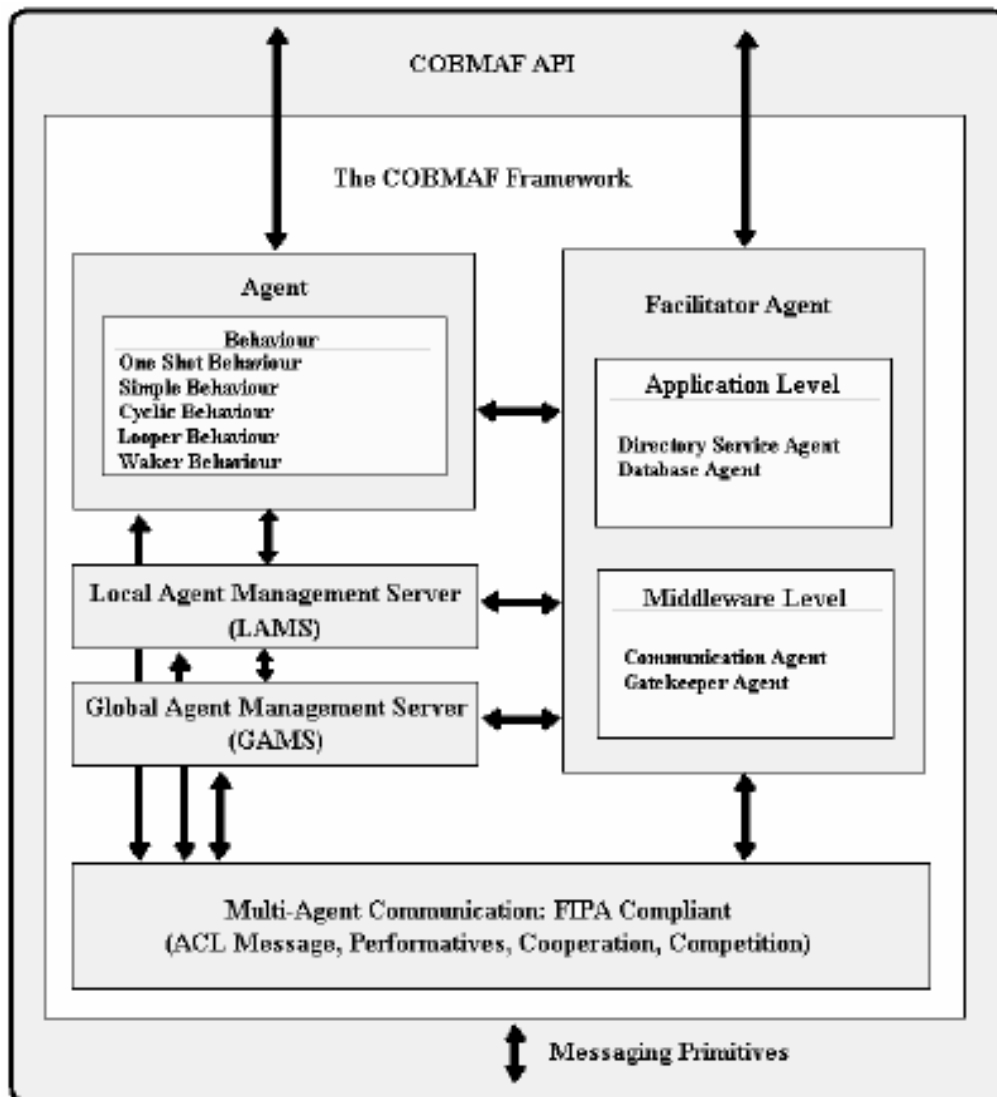


Fig. 1.3.2.2: COBMAF : Framework Layer and its components

2. COBMAF Version 2 – The Essential Upgrade

2.1 Overview

COBMAF version 2 is the upgrade to COBMAF version 1 and hence the name. COBMAF version 1 was successful in accomplishing the following objectives:

- I. Designing the structure of an agent based Virtual Enterprise system that is distributed and heterogeneous, with the focus on Automobile Manufacturing, along with the components, services and features available.
- II. Outlining the agent interaction scenarios and developing an ACL for communication between the agents with all the necessary performatives, the protocols and semantics for message passing and interaction between agents.
- III. Designing a negotiation algorithm for the agents involved.
- IV. Developing an underlying distributed message passing platform with all lower level primitives and semantics for agent communication in a heterogeneous environment - so CORBA was chosen as the underlying middleware.
- V. Developing a set of APIs that provided a framework for building the above designed system comprising agents with specialized behaviors and both, synchronous and asynchronous communication abilities, which forms the basis of the Virtual Enterprise.
- VI. Developing a set of standardized agents using the framework to demonstrate the working of a full-fledged Virtual Enterprise.

2.2 Problem Definition

COBMAF version 2 enhances COBMAF-version 1 and has the following sub goals in their respective order:

- I. Porting all the Basic Object Adapter (BOA) specific code to Portable Object Adapter (POA) code for extensibility and compatibility with newer versions of Visibroker (4.0 +) and JDK (Java 2 Platform) and thereby making COBMAF future ready to support upcoming releases of Visibroker and JDK
- II. Code compatibility with other popular ORBs such as Java 2 ORB, OpenORB, etc achieved so that the code can be used across different platforms and developer resources.
- III. Building a Directory Facilitator (DF) as a mandatory component of an Agent Platform that provides a yellow pages directory service to agents.
- IV. Building an ontological framework for COBMAF called COBONTO which is built upon the idea of standardizing agent communication.
- V. Upgrading the AWT to Swing API for performance and graphical upgrade.
- VI. Packaging the COBMAF Framework for ease of re-use to the developer.
- V. Building the COBMAF-WEB-MODEL (CWM) which is an altogether new framework for agent communication over the internet and borrows heavily from the CORBA IIOP architecture. The aim is to create 'Dynamically Interacting Enterprises' on the internet by utilizing the base classes of the COBMAF framework.

Thus COBMAF version 2 spearheads considerable enhancements in the directions of semantics, AI and standardization.

3. Porting from the Basic Object Adapter (BOA) to the Portable Object Adapter (POA)

3.1 What is the Portable Object Adapter (POA)?

An *object adapter* is the mechanism that connects a request using an object reference with the proper code to service that request. The Portable Object Adapter, or POA, is a particular type of object adapter that is defined by the CORBA specification. The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities.
- Provide support for transparent activation of objects.
- Allow a single servant to support multiple object identities simultaneously.

3.2 How the ORB uses POA?

The first thing that the ORB does is to locate the appropriate POA object. The POA is then to locate the responsible servant for the object for which the request is invoked. POAs are identified by name within the namespace of their parent POA. The location of POA starts from the root POA hierarchy (called rootPOA) and proceeds until the correct POA is found. The full path name is then extracted from the object reference to locate the position of the POA within the hierarchy. If the lookup of the POA hierarchy fails, then the application (programmer) has the opportunity to create and register the required POA by using an adaptor activator.

3.3 Why the need to port?

The existential code of COBMAF ver.1 was heavily depended on and constricted because of the use of Basic Object Adapter (BOA) which is the primary object adapter supported by Visibroker 3.x. If the COBMAF source code is to sustain upgrades to Visibroker 3.x i.e. Visibroker 4.x to Visibroker 6.x (current) then the BOA has to be ported to the POA.

With such a port, the COBMAF ver.2 release now supports not only Visibroker 6.x but also the latest releases of JDK 1.5 since this JDK version works only with Visibroker 4.x or later. COBMAF ver.2 is now ready to support any future releases of both Visibroker from Inprise and Java Development Kit (JDK) from Sun Microsystems. Moreover by changing only a few lines of the POA code, virtually any other ORB such as JavaORB, OpenORB can now be supported by COBMAF ver.2. Moreover, the most recent releases of Swing API from SUN can be fully exploited for richer Graphical User Interfaces within the COBMAF framework.

3.4 Guidelines for migration from BOA to POA

Client-side Changes

- Very little changes on the client side. Thus, many applications require no changes.

Server-side Changes

- POA_init is replaced with resolve_initial_references("RootPOA") followed by a _narrow operation.
- The implementation no longer inherits from the client-side stub. Instead, they inherit from PortableServer::ServantBase. The implications of this are (a) if you want a object reference for that, you must use the _this method.
- Object ID's are assigned by the POA unless you activate the servant with a specific ID.
- Unlike the BOA, the POA explicitly addresses the temporal nature of servants and declares that a POA can service either transient or persistent servants (not both). The root POA's (mandated, unchangeable) policy is "transient". The implications of this are that in order for a client to be able to manufacture an object reference on its own and use that to access an object, the servant for that object must be registered with a POA whose policy is "persistent". Thus, you

must create a child POA with that policy and register the servant with that POA. NOTE: when the POA declares something as "persistent", it is only stating that the key is valid between different runs of the server; it makes no claims that state or anything else is persistent.

- Servants are not automatically activated, hence you must register them by calling some of the activate_object methods on a POA or calling _this on the servant; with the latest you have no control on the ObjectId (which sometimes is good), and the POA must support the right policies (the RootPOA does).
- Servant constructors use to take a const char* parameter to set they object id, this is not needed now, in fact in many cases they use to pass this argument to the skeleton class: this will fail now.

3.5 How the porting was performed?

3.5.1 BOA code of COBMAF ver.1

```
org.omg.CORBA.ORB orbSales= org.omg.CORBA.ORB.init((String[])null,null);
org.omg.CORBA.BOA boaSales= ((org.omg.CORBA.ORB)orbSales).BOA_init();
myImpl implAMS= new myImpl(amsName);
boaSales.obj_is_ready(implAMS);
boaSales.impl_is_ready();
```

3.5.2 POA code of COBMAF ver.2

// Initialize the ORB.

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init((String[])null,null);
```

// get a reference to the root POA

```
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

// Create policies for our persistent POA

```
org.omg.CORBA.Policy[] policies =
```

```

{rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)};

// Create myPOA with the right policies
POA myPOA = rootPOA.create_POA(
"ams_poa",rootPOA.the_POAManager(),policies );

// Create the servant
myImpl amsServant = new myImpl();

// Decide on the ID for the servant
byte[] amsId = amsName.getBytes();

// Activate the servant with the ID on myPOA
myPOA.activate_object_with_id(amsId, amsServant);

// Activate the POA manager
rootPOA.the_POAManager().activate();
System.out.println(myPOA.servant_to_reference(amsServant) + " is ready.");

// Wait for incoming requests
orb.run();

```

Thus, all that is needed to migrate from BOA to POA is the replacement of the block of code in section 3.5.1 by code in section 3.5.2

4 Directory Facilitator

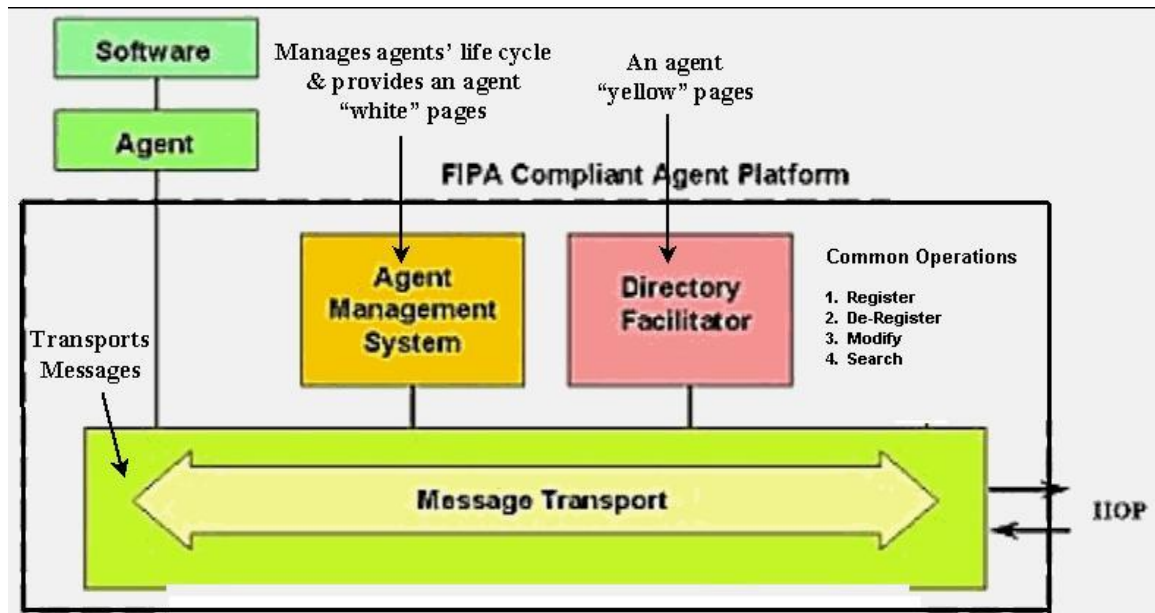
4.1 Overview

A DF is a mandatory component of an Agent Platform that provides a yellow pages directory service to agents. It is the trusted, benign custodian of the agent directory. It is trusted in the sense that it must strive to maintain an accurate, complete and timely list of agents. It is benign in the sense that it must provide the most current information about agents in its directory on a non-discriminatory basis to all authorised agents. At least one DF must be resident on each system (the default DF). However, the system may support any number of DFs and DFs may register with each other to form federations.

Every agent that wishes to publicise its services to other agents, should find an appropriate DF and request the **registration** of its agent description. There is no intended future commitment or obligation on the part of the registering agent implied in the act of registering. For example, an agent can refuse a request for a service which is advertised through a DF. The **deregistration** function has the consequence that there is no longer a commitment on behalf of the DF to broker information relating to that agent. At any time, and for any reason, the agent may request the DF to **modify** its agent description.

An agent may **search** in order to request information from a DF. The DF does not guarantee the validity of the information provided in response to a search request, since the DF does not place any restrictions on the information that can be registered with it.

4.2 Architecture – Where does the DF fit in the FIPA Agent Platform?



4.3 Management Functions Supported by the Directory Facilitator

In order to access the directory of agent descriptions managed by the DF, each DF must be able to perform the following functions, when defined on the domain of objects of type in compliance with the semantics described in section 4.1.4.1, *Directory Facilitator Agent Description*:

- `register` - The execution of this function has the effect of registering a new object into the knowledge base of the executing agent. See section 4.1.5.1 for function description.
- `deregister` - An agent may deregister an object in order to remove all of its parameters from a directory. See section 4.1.5.2 for function description.
- `modify` - An agent may make a modification in order to change its object registration with another agent. See section 4.1.5.3 for function description.

- **search** - The DF encompasses a search mechanism that searches first locally and then extends the search to other DFs, if allowed. See section 4.1.5.4 for function description.

Implementation key: In order to implement all the above services, a multi-dimensional array is used which will persist during the entire life-cycle of the DF.

4.4 DF Object Descriptions

The following terms are used to describe the objects of the domain:

- **Frame:** This is the mandatory name of this entity, which must be used to represent each instance of this class.
- **Ontology:** This is the name of the ontology, whose domain of discourse includes the parameters described in the table.
- **Parameter:** This is the mandatory name of a parameter of this frame.
- **Description:** This is a natural language description of the semantics of each parameter.
- **Type:** This is the type of the values of the parameter: Integer, Word, String, etc.
- **Reserved Values:** This is a list of FIPA-defined constants that can assume values for this parameter.

4.4.1 Directory Facilitator Agent Description

This type of object represents the description that can be registered with the DF yellow-page service.

Frame	DFAgentDescription		
Ontology	DFOntology		
Parameter	Description	Type	Reserved Values
name	The identifier of the agent.	String	
loc	The location of the agent	String	
services	A list of services supported by this agent.	Set of ServiceDescription	
protocol	A list of interaction protocols supported by the agent.	Set of String	See [FIPA00025]
ontology	A list of ontologies known by the agent.	Set of String	DFOntology
language	A list of content languages known by the agent.	Set of String	FIPA-SL CSL-C

4.4.2 Service Description

This type of object represents the description of each service registered with the DF.

Frame	ServiceDescription		
Ontology	DFOntology		
Parameter	Description	Type	Reserved Values
name	The name of the service.	String	
type	The type of the service.	String	Developer Defined
protocol	A list of interaction protocols supported by the service.	Set of String	
ontology	A list of ontologies supported by the service.	Set of String	DFOntology
language	A list of content languages supported by the service.	Set of String	FIPA-SL CSL-C
ownership	The owner of the service	String	
properties	A list of properties that discriminate the service.	Set of String	Developer Defined (Optional)

4.5 DF Function Descriptions

The following terms are used to describe the functions of the domain:

- **Function:** This is the symbol that identifies the function in the ontology.
- **Ontology:** This is the name of the ontology, whose domain of discourse includes the function described in the table.
- **Supported by:** This is the type of agent that supports this function.
- **Description:** This is a natural language description of the semantics of the function.
- **Domain:** This indicates the domain over which the function is defined. The arguments passed to the function must belong to the set identified by the domain.
- **Range:** This indicates the range to which the function maps the symbols of the domain. The result of the function is a symbol belonging to the set identified by the range.
- **Arity:** This indicates the number of arguments that a function takes. If a function can take an arbitrary number of arguments, then its arity is undefined.

4.5.1 Registration of an Object with the DF

Function	register
Ontology	DFOntology
Supported by	DF
Description	The execution of this function has the effect of registering a new object into the knowledge base of the executing agent. The DF acts as the executing agent when agents like RMA, Scheduler, Procurement, etc. register with it.
Domain	DFAgentDescription-Domain
Range	The execution of this function results in a change of the state, but it has no explicit result. Therefore there is no range set.
Arity	1

4.5.2 Deregistration of an Object with the DF

Function	deregister
Ontology	DFOntology
Supported by	DF
Description	An agent may deregister an object in order to remove all of its parameters from a directory. The DF acts as the executing agent.
Domain	DFAgentDescription-Domain
Range	The execution of this function results in a change of the state, but it has no explicit result. Therefore there is no range set.
Arity	1

4.5.3 Modification of an Object Registration with the DF

Function	modify
Ontology	DFOntology
Supported by	DF
Description	An agent may make a modification in order to change its object registration with another agent. The argument of a modify function will replace the existing object description stored within the executing agent. The DF acts as the executing agent.
Domain	DFAgentDescription-Domain
Range	The execution of this function results in a change of the state, but it has no explicit result. Therefore there is no range set.
Arity	1

4.5.4 Search for an Object Registration with the DF

Function	search
Ontology	DFOntology
Supported by	DF
Description	An agent may search for an object template in order to request information from an agent, in particular from a DF. A successful search can return one or more agent descriptions that satisfy the search criteria and a null set is returned where no agent entries satisfy the search criteria. The DF acts as the executing agent.
Domain	DFAgentDescription-Domain
Range	Set of objects. In particular, a set of DFAgentDescription (for the DF)
Arity	2

4.6 Implementing the DF

Concept: First define **Services** object, then encapsulate it in **DFAgentDescription** object and finally encapsulate use the DFAgentDescription object to set parameters of ACLMessage. Once this has been done the sendACLMessage (ACLMessage aclmsg) function can be invoked to send a registration message to the DF.

DFAgentDescription: set of {

Name: String
Location: String
Protocols: set of String
Ontologies: set of String
Languages: set of String

Services: set of {

Name: String
Type: String
Protocols: set of String
Ontologies: set of String
Languages: set of String
Ownership: set of String
Properties: set of String
}

}

Code Snippet showing how the implementation takes place:

```
If (Count !=0){  
int j=0;  
for (; j<Count; j++)  
{
```

Step1: *Instantiate a **ServiceDescription** object:: **serviceDescr** and set its parameters*

*Note: All parameters of the **ServiceDescription** object are not required to be set and solely depend on developer discretion.*

```
ServiceDescription serviceDescr = new ServiceDescription();
serviceDescr.setName(array1[j]);
serviceDescr.setType(agentType);
serviceDescr.addProtocols(FIPA-ContractNet-Protocol);
serviceDescr.addOntologies(df_ontology);
serviceDescr.addLanguages(CSL-C);

try{
InetAddress me = InetAddress.getLocalHost ();
OwnerIP = me.getHostAddress();
}catch(Exception e){System.out.println("Error in obtaining self IP Address");}

serviceDescr.setOwnership(OwnerIP);
```

Step 2: *Instantiate an **DFAgentDescription** object:: **dFAgentDescription** and encapsulate object:: **serviceDescr** into it*

*Note: All parameters of the **DFAgentDescription** object are not required to be set and solely depend on developer discretion.*

```
DFAgentDescription dFAgentDescription = new DFAgentDescription();
dFAgentDescription.setName(agentName);
dFAgentDescription.setLoc(OwnerIP);
dFAgentDescription.addServices(serviceDescr);
```

Step 3: *Instantiate an **ACLMessage** object:: **aclmsg1** and use the **dFAgentDescription** object parameters to initialize it*

```
ACLMessage aclmsg1=new ACLMessage("REGISTER");
```

```
aclmsg1.addServices(dFAgentDescription.services[j].getName());
```

```
} // End of inner FOR loop
```

```
aclmsg1.setOwner(dFAgentDescription.services[0].getType());
```

```
aclmsg1.setOwnerLoc(dFAgentDescription.getLoc());
```

```
aclmsg1.setContent("Register with DF");
```

```
aclmsg1.setLanguage("CSL-C");
```

```
aclmsg1.setOntology(df_ontology.getName());
```

```
aclmsg1.addReceiver("DF-Machine1");
```

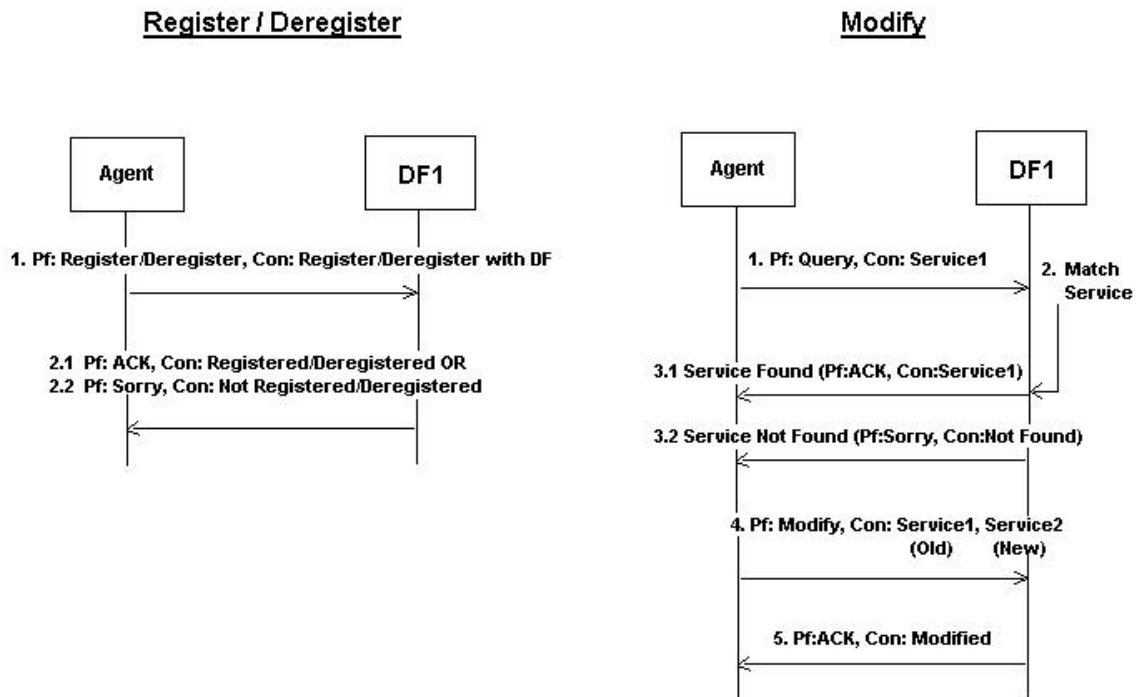
Step 4: Invoke **sendACLMessage (ACLMessage aclmsg)** function to register with DF

```
sendACLMessage(aclmsg1);
```

```
} // End of outer IF Loop
```

4.7 Agent-DF & Agent-DF-DF-Agent Interaction Diagrams

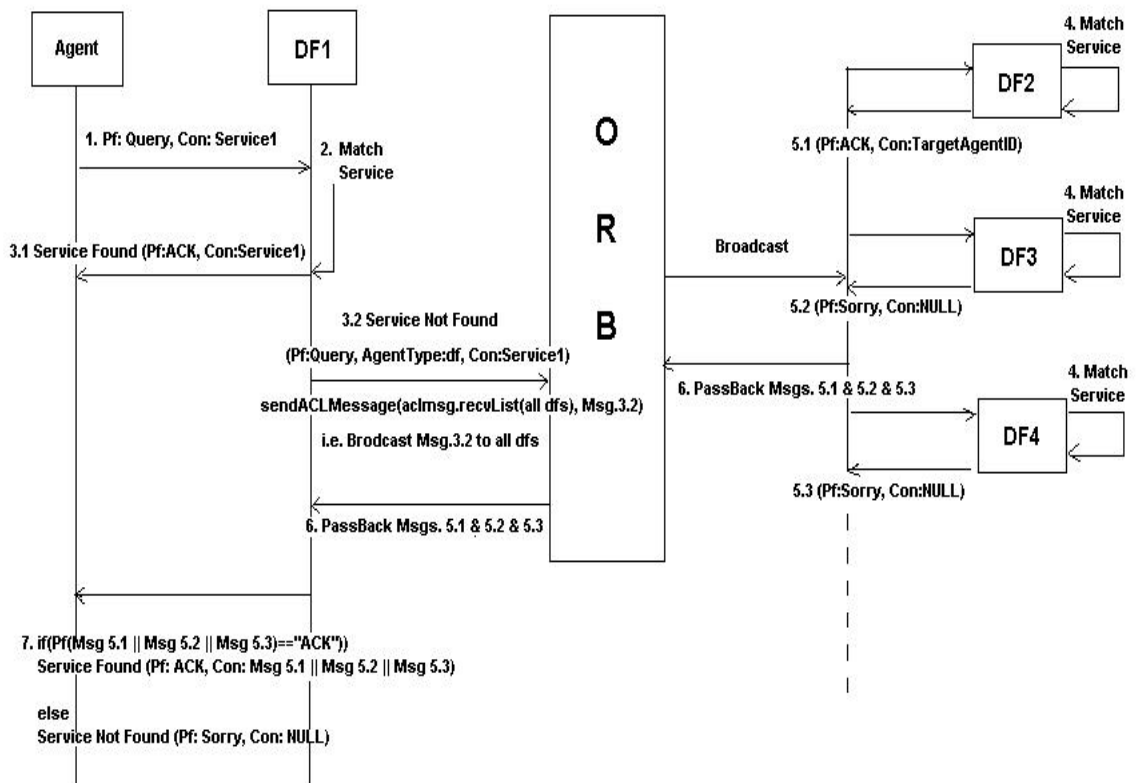
4.7.1 Register / Deregister & Modify Interaction Diagrams



The registration / de-registration of agents with DF initiates with the Agent sending an ACLMessage to DF1 with the performative as Register or De-Register and the Content as Register / Deregister with DF. Once this message has been received by DF1 it accordingly will send an ACK or Sorry Message back to the Agent confirming the success or failure of its Registration / Deregistration request.

The modification of services by DF1 begins with the Agent sending a Query message indicating the service that it wishes to modify. If the service is found then an ACK message is sent back and the Agent now proceeds by sending a Modify message to DF1 indicating the new Service name that it wishes to get registered with DF1. If the service is not found on initial query then a Sorry message is sent back to the Agent by DF1 and the Agent cannot proceed by sending a Modify message. Instead it is forced to query DF1 again

4.7.2 Search Interaction Diagram



The search function is a bit more complicated since this requires that the complete cycle of Agent-DF-DF-Agent be completed. The search for a particular service on the Local DF1 is exactly similar to the query phase of Modify service as detailed in section 4.6.1. However, the next phase is different. Instead of directly sending a Sorry Message, DF1 now sends a Query message on the ORB indicating the Service name and the agent type as df. This broadcast message is sent by utilizing the `addreceiver()` function of `ACLMessage` class and thereby generating a receiver list containing all agents of type df. The message is then sent on the ORB and reaches each of the other DF agents i.e. DF2, DF3, DF4, etc. which have already been registered on the ORB with ids df2, df3, df4, etc. When this message is received by each of these DFs a matching search is performed on the local database of these DFs. If the particular service is found then the ID of the target agent to which that service belongs is returned. Remember that the ID of all agents on the ORB is unique and hence by getting a reference to a particular agent's ID it can be directly referenced in the future by the caller Agent. Now if either of the messages returned by all these DFs is

positive, then the Agent has obtained the service and ID of the agent on the network to which it belongs and the Agent can now reference the target agent. If all the messages by these DFs are negative, then the Agent has to wait for some time and query again or query over IIOP to a different ORB i.e. different network.

5. ONTOLOGY

5.1 Overview

Definition: Ontology is a specification of a conceptualization of a knowledge domain. An ontology is a controlled vocabulary that describes objects and the relations between them in a formal way, and has a grammar for using the vocabulary terms to express something meaningful within a specified domain of interest. The vocabulary is used to make queries and assertions. Ontological commitments are agreements to use the vocabulary in a consistent way for knowledge sharing

COBMAF specific ontology: COBONTO is an ontological framework for COBMAF. COBONTO is built upon the idea of standardizing agent communication within the domains of:

1. The context of communication (ontology)
2. The coding-decoding (CODEC) scheme
3. Content language
4. The actual content, as parameters to the messaging system used to facilitate agent interaction.

COBONTO will support a number of Agent Communication Languages (ACLs), Content Languages and Application Specific Ontologies to give developers of the COBMAF framework maximum flexibility in terms of method of implementation. Creation of custom ontologies is discussed at length and a Virtual Enterprise (VE) specific Content Language called CSL-C is introduced.

5.2 Introduction

Agent architectures need both representation and communication models. Agent representation models include *ontologies* that define the domain model/vocabulary etc. of a particular domain of discourse, and *content languages* that represent the agent's mental model of the world (e.g. beliefs, desires, intentions). Given a particular domain of discourse, and a particular community of agents that know and do something in this domain, one needs an agent communication model that models the flow of knowledge and attitudes about such knowledge within the agent community. An ACL (agent communication language) provides language primitives that implement the agent communication model. ACLs are commonly thought of as *wrapper languages* in that they implement a knowledge-level communication protocol that is unaware of the choice of content language and ontology specification mechanism.

5.3 Rationale

When an agent A communicates with another agent B, a certain amount of information *I* is transferred from A to B by means of an ACL message (FIPA-ACL, KQML, CSL etc). Inside the ACL message, *I* is represented as a content expression consistent with a proper content language (e.g. FIPA-SL, KIF) and encoded in a proper format (e.g. string). Both A and B have their own (possibly different) way of internally representing *I*. Taking into account that the way an agent internally represents a piece information must allow an easy handling of that piece of information, it is quite clear that the representation used in an ACL content expression is not suitable for the inside of an agent.

For example the information that *there is a person whose name is Giovanni and who is 33 years old* in an ACL content expression could be represented as the string

(Person :name Giovanni :age 33)

Storing this information inside an agent simply as a string variable is not suitable to handle the information as e.g. getting the age of Giovanni would require each time to parse the string. Considering software agents written in C++ / Java (as COBMAF agents are), information can conveniently be represented inside an agent as C++ / Java objects. For example representing the above information about Giovanni as an instance (a C++ / Java object) of an application-specific class would allow handling it very easily.

```
Class Person {  
String name;  
int age;  
public String getName () {return name ;}  
public void setName (String n) {name = n ;}  
public int getAge () {return age ;}  
public void setAge (int a) {age = a ;}  
....  
}  
initialized with name = "Giovanni"; age = 33;
```

It is clear however that, if on the one hand information handling inside an agent is eased, on the other hand each time agent A sends a piece of information *I* to agent B,

1) A needs to convert his internal representation of *I* into the corresponding ACL content expression representation and B needs to perform the opposite conversion.

2) Moreover B should also perform a number of semantic checks to verify that *I* is meaningful piece of information, i.e. both A and B ascribe a proper meaning to *I*.

5.4 COBONTO - The Ontological Framework

Ontology in COBMAF is an instance of the `cobmaf.content.onto.Ontology` class to which the schemas defining the structure of the types of predicates, agent actions and concepts relevant to the addressed domain can be added by the developers.

Ontology implementation within the COBMAF-VE framework has been segregated according to the 2 primary phases that are involved in agent communication: Setup & Trade (Negotiation) (See section .C). Ontological support within COBMAF is achieved through a set of classes, messaging framework and implementation structure as follows:

- A) Base Class and its derivatives**
- B) Message Exchange Framework (MEF)**
- C) Implementation**

A) Base Class and its derivatives

A.1. `cobmaf.content.onto.Ontology` class:

The `Ontology` class is the base class within the COBMAF framework. Other ontologies are built on top of this class through the inheritance model i.e. by simply extending it. E.g. `public class DFOntology extends Ontology`. These are called derivatives and custom ontologies are built using this.

- This class implements the `java.io.Serializable` class so that the ontologies derived from it can be wrapped as serializable java objects and as parameters within the `sendACLMessage()` and `recvACLMessage()` functions.

- It contains the following 3 constructors:

- a. **`Ontology ()`** – Default constructor to create an unnamed instance of the `Ontology` class

b. **Ontology (String name)** – Constructor used to create a named instance of the Ontology class with the desired name of the derived ontology passed as a string parameter

c. **Ontology (String name, Ontology base)** – Constructor used to create a named instance of the Ontology class from a given base class with the desired name of the derived and base ontologies passed as string parameters

In addition to the above 3 constructors, 2 other functions are provided:

d. **Ontology getInstance () {return theInstance ;}** – This function allows the system to dynamically create and return an instance of the desired ontology class. For e.g. `Ontology setup_ontology = DFOntology.getInstance()` where `DFOntology` is a custom ontology class provided with the COBMAF-VE framework

e. **String getName () {return OntoName ;}** – This function returns the name of the target Ontology. For e.g. `String onto = DFOntology.getName ()`. Here the value present in private `String OntoName` is returned. It is most commonly used when an `ACLMessage` is to be sent by one agent. For e.g. `aclmsg1.setOntology (trade_ontology.getName ())`. Here the ontology variable of `ACLMessage` is set to the value “Trade-Ontology” as returned by the function: `trade_ontology.getName ()`.

A.2. cobmaf.content.onto.DFOntology class:

The `DFOntology` class provides the required semantic framework during 'Setup Phase' and builds on top of the performatives marked `*setuppfs*` in `ACLMessage.java`. The following diagram illustrates the agent interaction during 'Setup Phase'

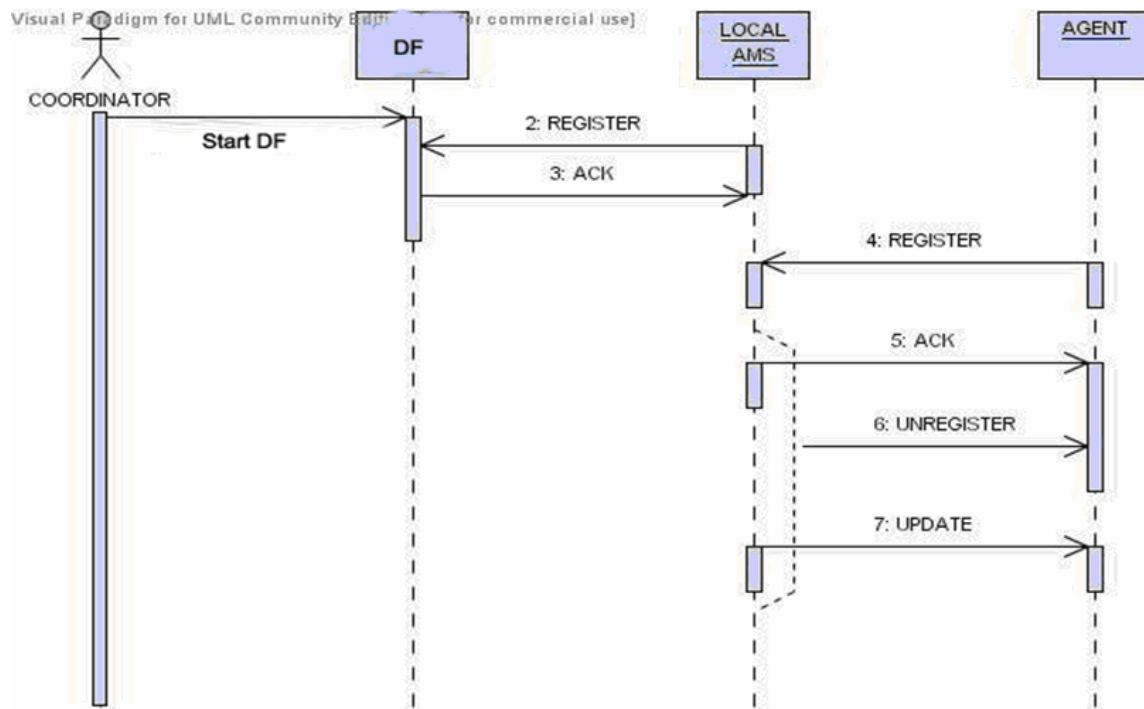


Figure 1.1

A.2.1. Setup-Phase Ontology (uses DFontology):

As seen in the above figure the setup-phase is initiated with the LocalAMS sending a 'REGISTER' performative to the DF which is available through the CSL ACL. This performative gives information of the intent of the message but says very little about how the registration is to be accomplished. Moreover there isn't a semantic basis or more appropriately any ontology to realize the context of the ensuing interaction. For the interaction to be unequivocal and the message to be clearly interpreted by both the sender and receiver agents, an instance of the DFontology class which is derived from the base Ontology class has to be dynamically created by the system at runtime so that the name of this ontology can be passed as one of the parameters to the ACLMessage which acts as the wrapper, implemented through the sendACLMessage() and recvACLMessage() functions. The following lines of code give a clear idea of how an object of DFontology is created and passed between the sender and receiver agents through Java serialization and the underlying CORBA framework:

A.2.2. LocalAMS to DF Message passing:

```
ACLMessage aclmsg = new ACLMessage("REGISTER");  
aclmsg.setContent("Register with DF");  
aclmsg.setLanguage("CSL");  
aclmsg.setOntology(SetupOntology.getName());  
aclmsg.addReceiver("DF");  
sendACLMessage(aclmessage);
```

Firstly, an instance of the ACLMessage class is created and instantiated with the performative "REGISTER" which is the main intent of the message that the LocalAMS agent want to send to the DF. Then the actual content is set and an associated Codec (Coding-Decoding) scheme is utilized to send the actual message to the DF. However each Codec has an associated Content Language and so the name of the content language used by the developer is specified. The next line of code is the most critical of the entire block since it indirectly initializes and specifies an object of the class DFOntology. The name Setup-Ontology is returned and acts as the basis for the entire message content. Without this the sender and receiver agents will fail to realize the context or the domain of vocabulary within which performatives like REGISTER are being used.

A.2.3. DF to LocalAMS Message passing and interpretation:

```
ACLMessage aclmsg = recvACLMessage();  
String OntoName = aclmsg.getOntology();  
String Language = aclmsg.getLanguage();  
String Content = aclmsg.getContent();
```

At the receiver side exactly the reverse process as that at the sender is carried out. Firstly, an instance of the ACLMessage class is initialized with the value returned by the function recvACLMessage(). Then the names of ontology, language and content are retrieved from the associated functions of the ACLMessage class. Now the DF is able to decode the entire content of the message i.e. the context (ontology) of message

communication, the language used for coding the actual content and the content in the form of strings. Since the entire message protocol and content are clear, the DF can now return an acknowledgement message on the same lines as that performed by the LocalAMS. Thus there is no confusion in agent communication at the semantic level and ease of messaging with the added security of a codec scheme is accomplished by the system.

A.2.4. Structure of cobmaf.content.onto.DFOntology class

```
public class DFOntology extends Ontology {
public static final String ONTOLOGY_NAME = "DF-Ontology";

// The singleton instance of this ontology
private static Ontology instance = new DFOntology ();

// Method to access the singleton ontology object
public static Ontology getInstance () {return instance;}

// Private constructor
private DFOntology() {
super (ONTOLOGY_NAME, Ontology.getInstance ());

/*Template for Developer to extend DFOntology
*1.COBMAF-VE does not require this block
*2.COBMAF-XXX may require this block
*3.Support for ConceptSchemas, AgentActionsSchemas will have to be added by
developer if 3 holds true.

*try {

// ----- Add Concepts

// ----- Add AgentActions
catch (OntologyException oe) {
```

```

oe.printStackTrace ();
}*/
}
} // DFOntology

```

A.3. cobmaf.content.onto.TradeOntology class:

The TradeOntology class provides the required semantic framework for actual negotiation during 'Agent Interaction Phases' and builds on top of the performatives marked *tradeperms* in ACLMessage.java

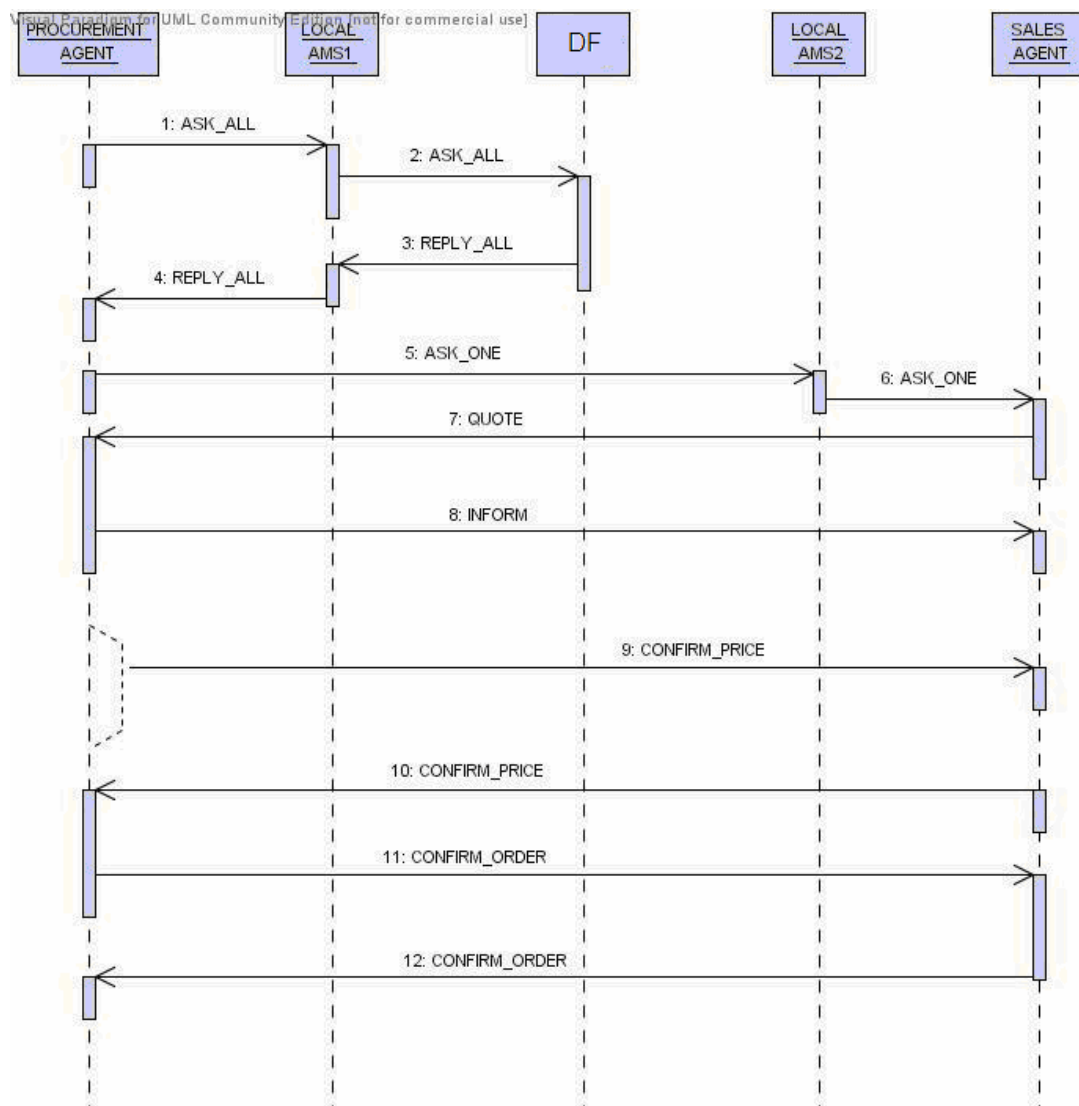


Figure 1.2

A.3.1. Trade (Negotiation) - Phase Ontology:

The message communication within this phase is based on the same lines as that of the Setup-Phase Ontology except that the sender and receiver agents are now the different agents such as Sales and Procurement agents. The communication between them now uses the Trade-Ontology as a context for message passing and interpretation. The developer is free to use the pre-installed Trade-Ontology or define his own ontology using the provided XXXOntology class for negotiation purposes. The performatives passed in this case include CONFIRM_PRICE, CONFIRM_ORDER, SCHEDULE etc while the content language remains CSL although the developer is free to use FIPA-SL or KIF if the need for additional performatives is felt or create his own and include them as part of the CSL-C Language.

A.4. cobmaf.content.onto.XXXOntology class:

The XXXOntology class acts as a general reusable template for the COBMAF application developer so which can be used to derive custom ontologies. Custom ontology classes can be created for COBMAF-XXX applications by the developer simply through the extension of base class Ontology and by making use of XXXOntology.java as template. The structural composition of Trade-Ontology class is exactly similar to that of the Setup-Ontology class.

B) Message Exchange Framework (MEF):

The MEF specifies a framework over which message exchange can be accomplished and consists of the following 3 primary components:

- 1. Set of Agent Communication Languages (ACLs)**
- 2. Set of Performatives**
- 3. Set of Content Languages**

B.1. Set of Agent Communication Languages (ACLs)

B.1.1. FIPA ACL – cobmaf.lang.acl.ACLMessage

FIPA ACL is the agent communication language associated with FIPA's open agent architecture. As with KQML, FIPA-ACL maintains orthogonality with the content language and is designed to work with any content language and any ontology specification approach

B.1.2. KQML – cobmaf.lang.kqml.KQMLMessage

KQML is the ACL resulting from the DARPA KSE (Knowledge Sharing Effort) effort, which also produced *KIF* as the content language (first-order logic + set theory) and *Ontolingua* as the ontology specification language. In keeping with the wrapper philosophy of ACLs, KQML is insensitive to whether the content it is communicating about is in KIF/Ontolingua or something else.

B.1.3. COBMAF Semantic Language (CSL) – cobmaf.lang.csl.CSLMessage

COBMAF Semantic Language (CSL) was initially created with the aim of creating a set of performatives and communication protocol applicable to our specific COBMAF-VAME (Virtual Automobile Manufacturing Enterprise) framework. However it was soon felt that the number of applications in the Virtual Enterprise (VE) sector is extremely high in and hence our research team found the need to create a semantic language targeted specifically to VE applications. Most of the performatives which are a part of CSL can be easily used in transaction-oriented and seller-buyer type of environments where the need to communicate in a transaction-based protocol is greatly felt. Some examples of these performatives include: CHECKORDER, CONFIRMPRICE, CONFIRMORDER, SCHEDULE, QUOTE etc. Current research is underway in standardizing a much larger set of performatives which can support diverse applications with the VE sector.

B.2. Supported set of Performatives – cobmaf.lang.xxx.XXXMessage

Currently 21 performatives of the popular FIPA-ACL language, 43 performatives of KQML language and 18 performatives of our CSL language are supported by the COBMAF framework. The idea is to support a larger pool of performatives so that many more diverse applications apart from VE can be supported by COBMAF.

B.3. Supported set of Content Languages – cobmaf.content.lang.xxx

COBMAF currently supports FIPA SL (SL0, SL1, SL2) and KIF (Knowledge Information Exchange) as general purpose Content Languages (CLs) and CSL-C as a VE-Application specific CL. As the integration of more ACLs is accomplished, so will the parallel inclusion of different content languages. The purpose of such a wide support within the COBMAF framework is to be able to support more number of application developers who are proficient within the domain of a given ACL or CL and want to take advantage of the superior features of the CORBA-based COBMAF framework.

C) Implementation:

The actual implementation of ontology within the COBMAF framework is a group of four parameters passed to the ACLMessage object. They are the performative (intent), content, content language and ontology (context). Each of these 4 parameters has to be passed to abide to the flexible COBMAF-Ontological Framework. This framework is open enough to support a number of different content languages and ontologies either provided as part of COBMAF or custom built by the developers for specific purposes. The following lines of code illustrate exactly how the implementation is accomplished within the different agent java files:

```
ACLMessage aclmsg=new ACLMessage("Performative Name");  
aclmsg.setContent("Content String");  
aclmsg.setLanguage("Content Language");  
aclmsg.setOntology("Ontology Name");
```

```
aclmsg.addReceiver("Receiver Agent");  
sendACLMessage(aclmessage);
```

Example -

```
ACLMessage aclmsg=new ACLMessage("REGISTER");  
aclmsg.setContent("Register with DF");  
aclmsg.setLanguage("CSL");  
aclmsg.setOntology(trade_ontology.getName());  
aclmsg.addReceiver(salesName);  
sendACLMessage(aclmessage);
```

These lines of code are the basis of setting and retrieving the context of the message and are spread throughout the COBMAF source code.

5.4) Future:

- A) Support for increased number of Agent Communication Languages, Performatives and Content Languages to facilitate greater developer flexibility, choice and heterogeneous agent interaction.
- B) Development of a universal semantic language called the COBMAF Semantic Language (CSL) to enable cross-lingual agent interaction. This can be achieved by developing a common set of performatives and content languages between agents abiding to different ACLs.

5.5) Screenshots:

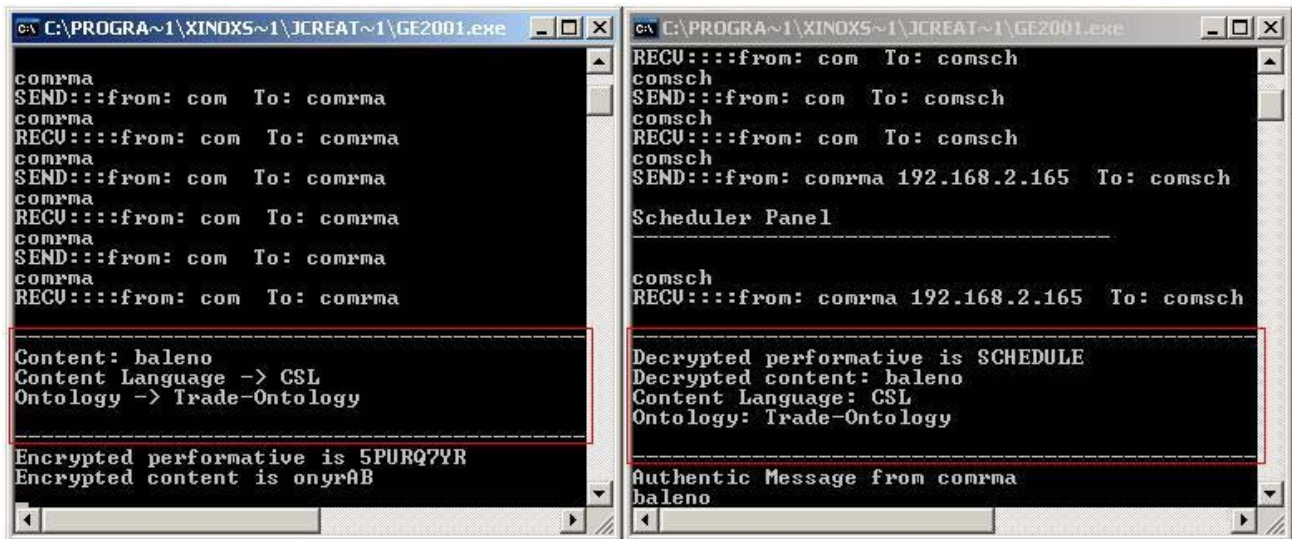


Figure 1.3 – RMA / Scheduler Agent Interaction using Trade-Ontology

6. THE COBMAF-SWING GRAPHICAL USER INTERFACE (CSGUI)

6.1 What is the CSGUI?

The COBMAF-Swing Graphical User Interface (CSGUI) of COBMAF ver.2 is a significant upgrade to the GUI employed in COBMAF ver.1.

6.2 Why the need to upgrade?

The GUI of COBMAF ver.1 relied heavily on the Java AWT (Abstract Window Toolkit) which lacked several features of the more recent Swing API released by SUN Microsystems. Also, porting the code to POA forced the development team to use JDK 1.5 which meant taking advantage of the Swing APIs directly found within this JDK version. Moreover, Swing readiness means that the COBMAF code is ready to work with future GUI releases by SUN and the look and feel of the system is also a lot more compelling and attractive. The section 6.3 gives a brief overview the pros and cons of AWT and Swing which led to the team's selection of Swing API as the primary GUI API on which COBMAF ver.2's graphical interface would be designed

6.3 AWT versus Swing

AWT:

Pros : Applet Portability: most Web browsers support AWT classes so AWT applets can run without the Java plug-in.

Cons

- Portability: use of native peers creates platform specific limitations. Some components may not function at all on some platforms.
- Speed: The AWT components did not suit the speed and memory usage standards of COBMAF ver.2
- Features: AWT components do not support features like icons and tool-tips.

Swing:

Pros

- **Portability:** Pure Java design provides for fewer platform specific limitations.
- **Speed:** Although Swing components are traditionally considered to consume greater processing power, there was a significant gain in the overall speed and memory usage at runtime which were primary drivers for choosing Swing as COBMAF's primary GUI API.
- **Look and Feel:** The pluggable look and feel lets you design a single set of GUI components that can automatically have the look and feel of any OS platform (Microsoft Windows, Solaris, Macintosh, etc.). It also makes it easier to make global changes to your Java programs that provide greater accessibility (like picking a hi-contrast color scheme or changing all the fonts in all dialogs, etc.).

Cons

- **Applet Portability:** Most Web browsers do not include the Swing classes, so the Java plugin must be used.
- **Look and Feel:** Even when Swing components are set to use the look and feel of the OS they are run on, they may not look like their native counterparts.

6.4 Why Swing was chosen ?

Speed, Look and Feel and Simplicity of GUI coding were primary drivers in choosing Swing API over AWT. The system was compared head to toe considering all these factors both when implemented using AWT and Swing APIs

With regards to speed of the program, although Swing components are traditionally considered to consume greater processing power, there was a significant gain in the overall speed and memory usage at runtime. This advantage of speed over AWT came from the fact that the AWT API simply was not able to support the memory usage of the user's machine. Too many AWT frames had to be initiated and refreshed at regular intervals of time thereby affecting the speed drastically. With Swing, the

option of docking all these frames in the form of tabs became possible and hence vital screen refresh times were eliminated which boosted the overall performance.

With regards to Look and Feel, Swing came out dominant with the pluggable look and feel letting the team design a single set of GUI components that can automatically have the look and feel of any OS platform (Microsoft Windows, Solaris, Macintosh, etc.). Also Swing made it a lot easier to make global changes to our Java code that provided greater accessibility (like picking a hi-contrast color scheme or changing all the fonts in all dialogs, etc.)

Finally with regards to Simplicity, Swing clearly held the upper hand, since Swing code is much easier to write and maintain and there are ready-to-use Swing GUI designers available within IDEs such as JBuilder by which the entire frames can be visually designed from the ground-up.

** See section 7 for referring to the Swing and API contents of the `cobmaf.gui` package

7. THE COBMAF PACKAGE STRUCTURE (CPS)

7.1 Overview

The COBMAF ver.2 package is structured with the principle of clearly delineating the source code files and thereby providing a simple mechanism for derivative usage of the built classes by the developer. This package structure thereby facilitates simplicity of re-use and development.

COBMAF is packaged as a set of 8 independent packages which are as follows:

7.2 cobmaf.content.onto package:

7.2.1 Package description: The cobmaf.content.onto package contains all the Ontology specific classes.

7.2.2 Package contents:

a) *Ontology* – The base ontology class from which all other ontology specific classes such as *DFOntology*, *TradeOntology* and *XXXOntology* are derived.

b) *DFOntology* – This class is derived from the base *Ontology* class and specifies the set of Agent Actions and Concepts which form a part of the DF-DF and DF-Agent communication domain.

c) *TradeOntology* – This class is derived from the base *Ontology* class and specifies the set of Agent Actions and Concepts which form a part of Sales-Procurement Negotiation communication domain.

d) *XXXOntology* – This class is derived from the base *Ontology* class and provides a template for the developer to write his own set of Agent Actions and Concepts which can form a part of any other Agent-Agent communication domain.

7.3 cobmaf.core package

7.3.1 Package description: The cobmaf.core package contains all the core class files from AMS to DF. In other words, cobmaf.core package is really the heart of the development package.

7.3.2 Package contents:

- a) *Agent* – This is the basic Agent class which extends Thread class and provides functions of addBehaviour(), sendACLMessage() and recvACLMessage().
- b) *AgentManager* – This class is what gets executed when the AMS is started. It contains code to manage registration and life-cycle of all the COBMAF Agents.
- c) *AgentServer* – This class is used to start the Agent Server.
- d) *AMSListener* – This class is used to listen to incoming requests from COBMAF agents for the purpose of registration.
- e) *DF* – The Directory Facilitator class extends the base Agent class and contains all code pertaining to the implementation of the DF. It displays a DF Registration Dashboard for all the currently registered agents.
- f) *DFAgentDescription* – This class contains parameters for describing the COBMAF agents such as name, location as well as functions for adding/removing services, protocols, ontologies and languages.
- g) *PertCPM* – This class contains code which implements the CPM-PERT method.
- h) *ProcAgent* – This class extends the base Agent class and contains code pertaining to the services of the procurement agent.
- i) *RMA* – This class extends the base Agent class and contains code pertaining to the services of the RMA agent.
- j) *SalesAgent* – This class extends the base Agent class and contains code pertaining to the services of the Sales agent.

k) *Scheduler* – This class extends the base Agent class and contains code pertaining to the services of the Scheduler agent.

l) *ServiceDescription* – This class contains parameters for describing the services of the COBMAF agents such as name, type as well as functions for adding/removing protocols, ontologies, languages and ownership.

m) *Task* – This class contains code which implements the Task Scheduler.

7.4 cobmaf.core.behaviours package

7.4.1 Package description: The cobmaf.content.onto package contains all the Behaviour specific classes of COBMAF agents.

7.4.2 Package contents:

a) *Behaviour* – This class serves as the base class from which all other behaviour specific classes such as CyclicBehaviour, LooperBehaviour, etc are generated.

b) *CyclicBehaviour* – This class contains code that implements the cyclic behaviour of COBMAF agents.

c) *LooperBehaviour* – This class contains code that implements the looper behaviour of COBMAF agents.

d) *OneShotBehaviour* – This class contains code that implements the one-shot behaviour of COBMAF agents.

e) *SimpleBehaviour* – This class contains code that implements the cyclic simple behaviour of COBMAF agents.

f) *WakerBehaviour* – This class contains code that implements the cyclic waker behaviour of COBMAF agents.

7.5 cobmaf.gui package

7.5.1 Package description: The cobmaf.gui package contains all the Swing and AWT components (for backwards compatibility).

7.5.2 Package contents:

- a) *dfFrame1* - This frame is used to create the DF Agent itself. It takes as input the name of the DF and the IP address of the machine on which the DF agent is created.
- b) *DFRegisterFrame* – This frame lists the names and IP addresses of the agents registered with it and provides an option for deregistering them.
- c) *DFUpdateFrame* – This frame updates the REGISTER/DEREGISTER option as and when the agents are registered or deregistered by the platform user or operator.
- d) *DialogB* – This frame has been deprecated in COBMAF ver.2. The primary use of this simple dialog box was to give a confirmation message back to the user whether the agent has been registered successfully or not by the DF.
- e) *pNegotiationUpdateFrame* – This frame is used to update the procurement price and belongs to the procurement agent.
- f) *procFrame1* – This frame is used to create the procurement agent and invokes the frame *procFrame2* for inputting additional details.
- g) *procFrame2* – This frame is invoked by *procFrame1* to enter additional details for the procurement agent.
- h) *salesFrame1* – This frame is used to create the sales agent and invokes the frame *salesFrame2* for inputting additional details.
- i) *salesFrame2* – This frame is invoked by *salesFrame1* to enter additional details for the sales agent.

j) *sFrame1* – This frame is used to create the scheduling agent which asks for its name, company name and IP address of the creator’s machine.

k) *sNegotiationUpdateFrame* – This frame is used to update the sales price offered by the sales agent and belongs to the sales agent

7.6 cobmaf.lang.acl package

7.6.1 Package description: The cobmaf.lang.acl package contains all the ACLMessage specific classes.

7.6.2 Package contents:

a) *ACLCodec* – This class contains code which implements ACLMessage Coder-Decoder and provides functions for encrypting and decrypting both the content of the message and its performative.

b) *ACLMessage* – This class provides FIPA-ACLMessage functions for setting and getting performatives and content and adding/removing services and receivers.

7.7 cobmaf.lang.csl package

7.7.1 Package description: The cobmaf.content.onto package contains all the CSLMessage specific classes.

7.7.2 Package contents:

a) *CSLCodec* – This class contains code which implements CSLMessage Coder-Decoder and provides functions for encrypting and decrypting both the content of the message and its performative.

b) *CSLMessage* – This class provides *CSLMessage* functions for setting and getting performatives and content and adding/removing services and receivers.

7.8 cobmaf.lang.kql package

7.8.1 Package description: The *cobmaf.content.onto* package contains all the *KQLMessage* specific classes.

7.8.2 Package contents:

a) *KQLCodec* – This class contains code which implements *KQLMessage* Coder-Decoder and provides functions for encrypting and decrypting both the content of the message and its performative.

b) *KQLMessage* – This class provides *KQLMessage* functions for setting and getting performatives and content and adding/removing services and receivers.

7.9 myInterface package

7.9.1 Package description: The *cobmaf.content.onto* package contains all the *Interface* specific classes.

7.9.2 Package contents:

a) *_example_Interface1* – This class is an automatically generated example class for the given application.

b) *_Interface1ImplBase* – This abstract class is the server skeleton, providing basic CORBA functionality for the server. It implements the *Interface1.java* interface.

c) *_sk_Interface1* – This abstract class is the skeleton for *Interface1*.

- d) *_st_Interface1* – This abstract class is the stub for Interface1.
- e) *Interface1* – This abstract class completely defines Interface1.
- f) *Interface1Operations* – This class contains functional implementation of Interface1.
- g) *Interface1Helper* – This final class provides auxiliary functionality, notably the narrow method required to cast CORBA object references to their proper types.
- h) *Interface1Holder* – This final class holds a public instance member of type Hello. It provides operations for out and in arguments, which CORBA has but which do not map easily to Java's semantics.
- i) *message* – This class contains the parameters and functions of the message object.
- j) *messageHelper* – This final class provides auxiliary functionality, notably the narrow method required to cast CORBA object references to their proper types.
- k) *messageHolder* – This final class holds a public instance member of type Hello. It provides operations for out and inout arguments, which CORBA has but which do not map easily to Java's semantics.
- l) *myImpl* – This class contains the actual implementation of the myInterface package.

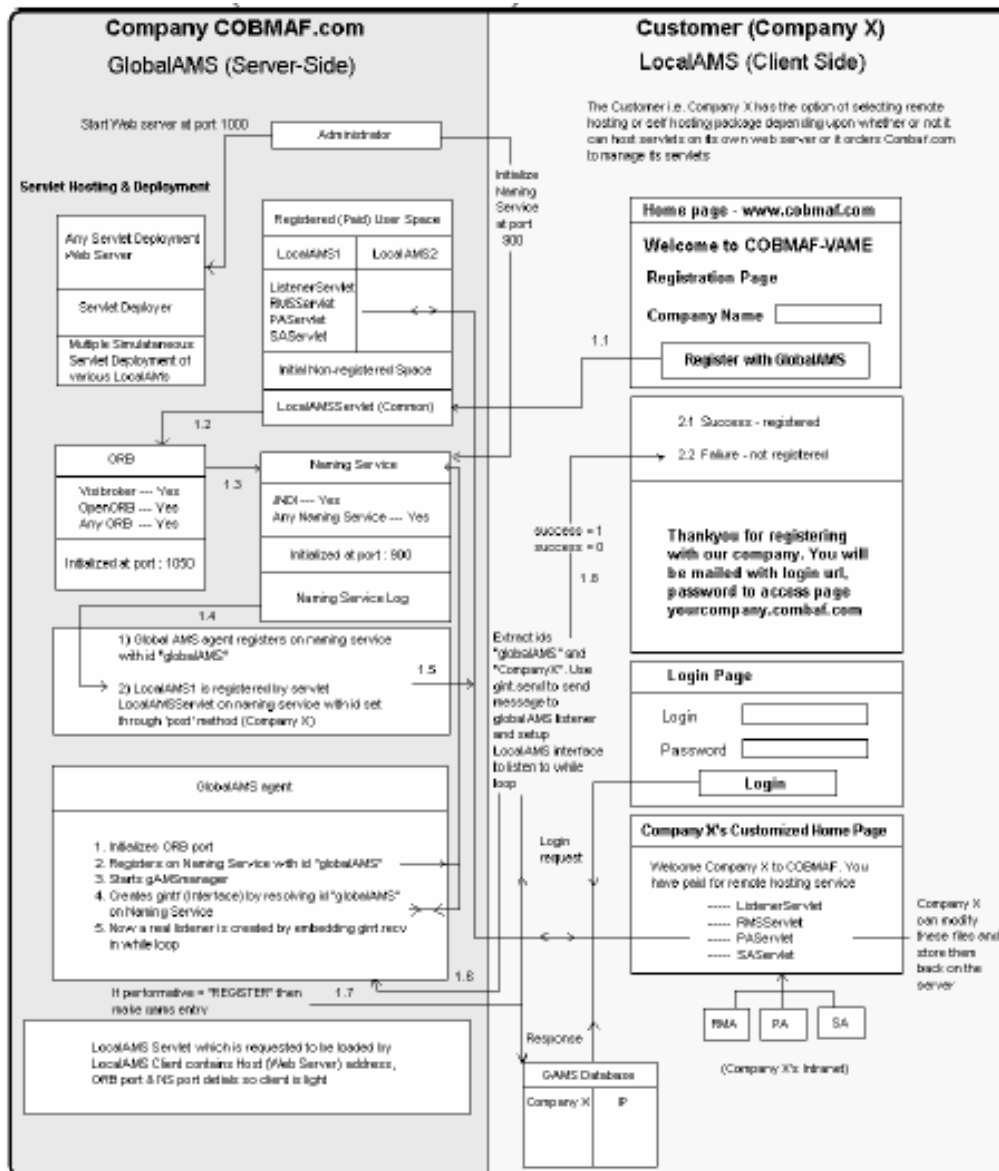
8. The COBMAF WEB MODEL (CWM)

8.1 Overview

The following proposed web model is aimed at mapping the COBMAF framework to the internet and thereby setting up an actual company called COBMAF.com which would provide a working implementation of our work to the masses. This web-model isn't a one-to-one mapping for our actual research work but has its own innovative and unique architecture. What is mapped though are the java classes (comprising of all agent related code) which my team wrote and the underlying CORBA architecture upon which our middleware is based. This no way is a replacement to our work but only a marketable deploy ready solution for Dynamically interacting Enterprises on the Internet.

Since this model was initially planned with the idea of developing standalone server/client or seller/buyer user interfaces, it can easily be mapped to the resources of the internet. Hence this marketable deploy-ready web-based model has been proposed.

8.2 Architecture



The architecture though complex at first sight is rather simple at its core with the basic idea being that there are in essence two interacting entities participating in the model. The first is the company COBMAF.com (setup by us) which would have all the agent related code which our team wrote and thereby acting at the server side. The second is any Company X (supplier or buyer) who wants to participate in the dynamic enterprise to locate other buyers and sellers. Hence such a Company X or LocalAMS (Local Agent Management System) is the customer for our parent company.

COBMAF.com which is also the DF (Directory Facilitator) as defined in our research work. The DF and all its server-side architecture and functioning is described in the next section.

8.3 The DF (Server-Side Architecture):

Servlet Hosting and Deployment:

The server-side architecture consists of any servlet enabled web server such as Apache Tomcat which would host and deploy all servlets related to the respective client companies such as Company X. More on how servlets are used is explained later. Each company who joins the dynamic enterprise is allotted its own registered user space where servlets corresponding to that company are stored on the web server.

The ORB and Naming Service:

Since our current research work makes all ORBs and Naming Services compatible to work with the Virtual Enterprise (this is made possible due to the middleware we have developed i.e. COBMAF) supporting different ORBs and Naming Services in the web-based model was a natural extension. The server-side DF therefore has the option of using various kinds of ORBs such as Visibroker, OpenORB and naming services such as JNDI and others.

Administrator:

The Administrator is any person in charge of setting up and managing the web server as well as initializing the Naming Service. The ORB is automatically initialized through the port values present in the code.

LocalAMSServlet (Common):

The LocalAMSServlet is common to all the LocalAMSs and acts as the intermediary between each LocalAMS and the DF. It registers each LocalAMS on the naming service.

DF agent:

The DF agent does the following things:

1) Initializes the ORB 2) Registers on the naming service with id "DF" 3) Starts the GlobalAgentManager which invokes another GlobalAgentListener class (agent) and handles registration of all LocalAMSs.

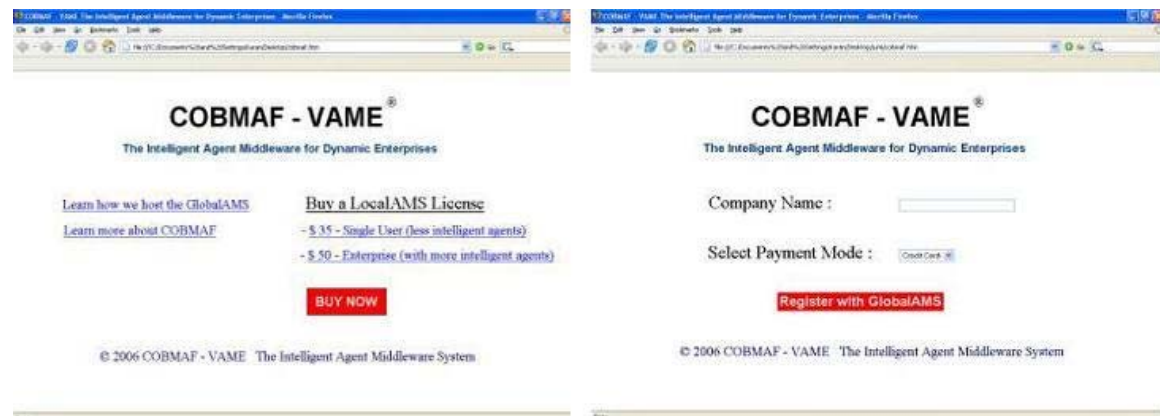
The use of Servlets:

The Servlets such as Listener Servlet, Resource Management Servlet, Procurement Servlet and Sales Servlet have been employed to implement all the agent related class files which our team wrote and making our standalone code compatible in the web world. Each of the above servlets are stored on the registered user space for each company and these servlets can be downloaded by the LocalAMSs so that they can tailor it to their exact preference and upload them back on the web server.

8.4 The LocalAMS (Client-Side Architecture):

Since all our core agent related class files have been migrated to the server side, the client side architecture is kept clean in the form of simple html pages which request and receive data through the servlet processing at the server side.

8.5 How our Web-Model works



The following procedure makes it clear to see how our Virtual Enterprise would actually function over the internet :

1) The Administrator at Company COBMAF.com first gets the servlet enabled web server up and running and manually initializes the Naming Service which is used by different participating companies to find each other and communicate.

2) Next the Administrator runs the DF class file (agent) which initializes the ORB and registers itself on the naming service with id "DF" and represents our company COBMAF.com. By registering itself this agent makes itself available to all other participating companies. It then automatically starts the GlobalAgentManager class file (agent) which invokes the GlobalAgentListener class and handles registration of all LocalAMSs in the df database containing entries in the form of LocalAMS name - IP. Now the server side startup is complete and our Company COBMAF.com is ready to listen to incoming registration requests from the client side.

3) Now the Company X or LocalAMS who wishes to participate in the Virtual Enterprise first visits our Company site: **www.cobmaf.com** (currently assumed to be fictitious) and is presented with a License buying screen as seen in Fig 1.1. When it clicks on Buy now, it is taken to the Registration page (Fig 1.2) where it enters its name and selects its mode of payment. Then it clicks on the button 'Register with DF' to send it registration request to the DF. This registration request is processed by the common LocalAMSServlet which finds the ORB and the Naming Service and

registers Company X on the Naming Service with id passed in the 'Company Name' field through the get and post methods.

4) Next the LocalAMSServlet extracts the ids DF and the recently registered "Company X" from the naming service. It then uses the available interface gint to send a 'REGISTER performative' to the listening DFListener and also sets up LocalAMS interface to listen to the corresponding response from the DF.

5) The DFAgentListener which is part of the GlobalAgentManager now received the performative "REGISTER" after which it makes an entry of the type Company X name - IP in the gams directory. This confirmation goes back to the LocalAMS listener as success = 1 or success = 0. If the company is registered it is then taken to a login screen else an error message is displayed.

6) Once the registration procedure is complete, the Company X logs into the system at its sub domain : **companyx.cobmaf.com**. If it has chosen the option of hosting its agent related servlet code on the remote side then it simply downloads the servlet files displayed on the screen, makes suitable changes to the code, and uploads the files.

7) A Company Y who is part of the seller-buyer network of Company X now acts like the LocalAMS and Company X acts like the DF and the procedure repeats.

9. CONCLUDING FEATURES AND TEST PARAMETERS

9.1 Transparency Issues

Definition: Transparency involves making the user to believe that the entire distributed system is in fact a single system. The various types of transparencies that need to be taken care of are:

Kind	Meaning
Location Transparency	The users cannot tell where the resources are located
Migration Transparency	Resources can move at will without changing their names.
Replication Transparency	The users cannot tell how many copies exist
Concurrency Transparency	Multiple users can share resources automatically
Parallelism Transparency	Activities can happen in parallel without users knowing.

Purpose: For the various organizations registering with COBMAF, it is necessary to maintain a high level of transparency to avoid exchange of unwanted information.

The COBMAF Solution: The agents communicate with each other via the ACL and the messages are passed to each other using the CORBA framework. The Visibroker Smart Agent which is used to provide the CORBA layer helps in making the communication completely transparent to the client program. The various transparency issues mentioned above are dealt by COBMAF in a manner as listed in the Test Scenarios.

Test Scenario:

- *Location Transparency:* The agents register with DF and advertise the services that can be provided by them. But at the same time the Agents can either accept or reject the services that have been advertised by the DF. The Agents can search for the services currently available by searching in the DF Yellow Page service. The services called for are then available to the Agents and they can use them without having to worry about the location of the Agent providing the service. But the Agent does have the knowledge about the Agent and the organization to which it belongs and the kind of ontology supported by it. Thus COBMAF adds to the location transparency as required by a distributed system.
- *Migration Transparency:* As the current adaptation of the COBMAF framework doesn't include Migration of Agents, this issue of Distributed Systems is not applicable over here.
- *Replication Transparency:* The COBMAF framework doesn't have any duplication of services for backup incase of failure. The Visibroker Smart Agent though has services such as load balancing and Naming Service (which is not used by COBMAF due to the introduction of the DF Agent). The Smart Agent locates the specified implementation so that a connection can be established between the client and the implementation. When a particular instance of a Smart Agent is terminated due to some uncalled for circumstances, then the activities of the Agents registered with it is passed on to another instance of Smart Agent running in the same network.
- *Concurrency & Parallelism Transparency:* Every Agent Class extends the Thread Class that ensures that every agent has a separate thread of control whose execution is operating system controlled. Thus concurrent accesses to the DF and the Local AMS are handled by independent threads thus handling the concurrency related issues of the Distributed system. The COBMAF framework also permits the parallel execution of task by use of threads. *E.g.*

An agent can carry out negotiations as well provide services to other agents at the same time due to the use of threads.

9.2 Scalability

- *No. of Organizations joining the VE:* As mentioned in the Concurrency & Parallelism Transparency issues, every Agent class extends the Thread class. So every instance of the Agent has its own independent thread running parallel with many other threads. Thus theoretically speaking the COBMAF framework can support any number of Local AMS registering with it. But during the practical implementation of the framework it was noticed that when more than 3 Local AMS were started on the same machine, the performance and the response times for the agents and the time taken for their negotiations increased to a great extent due to the large overheads involved in the use of Threads. The elimination of the Global AMS as the central auditing authority in the current edition of the COBMAF framework and the use of swing components has reduced the load to some extent, but still the system experiences slowing down of all the processes and sometime leading to Java exceptions. When the Local AMS are started over different computers in the same network, the framework works fine.

- *No. of Behaviors that an Agent can invoke:* The COBMAF framework supports the following Behaviours:-
 1. One Shot Behaviour
 2. Simple Behaviour
 3. Cyclic Behaviour
 4. Looper Behaviour
 5. Waker Behaviour

These behaviors have been described before in the documentation and have been left untouched in the current adaptation of COBMAF.

These are the behaviours that can be added on the current framework to enhance the functionalities and providing the agents with a larger skill set to work with.

ParallelBehaviour: controls a set of *children* behaviours that execute in parallel. The important thing is the termination condition: we can specify that the group terminates when **ALL** children are done, **N** children are done or **ANY** child is done.

SequentialBehaviour: this behaviour executes its children behaviours one after the other and terminates when the last child has ended.

We can add these Behaviors to the present list by using the **void** `addBehaviour(Behaviour)` function. We can also remove a behavior if need be by defining a **void** `removeBehavior(Behavior)` function.

- *Ease of joining and leaving a VE:* The COBMAF framework provides us with a facility to de-register an already registered agent through the frame that lists the agents that have registered. De-registering an agent leads to its name and its services being removed from the yellow page service of the Directory Facilitator and any Agent currently communicating with the agent is forced to terminate its transaction. Also de-registering of the DF leads to all Agents registered with it to get de-registered. An agent can join a VE by simply filling up the registration form and by listing the services that it would be able to provide.

While scaling this project to a bigger picture, we can introduce a Gatekeeper Agent which would enable the Agents to communicate and carry out transactions with Agents in other networks across firewalls. A distributed system involves communicating with entities which are protected behind a firewall and use different security features. The GateKeeper Agent is used to efficiently handle such diverse security restrictions while keeping the agents transparent to such details. It can use the gatekeeper functionality provided by VisiBroker for CORBA.

9.3 Failsafe / Self-configuration nature:

Definition: The failsafe / self-configuration nature of COBMAF is the ability of the framework to recover from unexpected failures or exceptions during the startup, runtime and shutdown modes and thereby account for varying degrees of fault-tolerance which are acceptable for achieving high performance results.

Purpose: To make the framework failsafe during either of the above mentioned phases and thereby reduce the risk of improper agent communication interruption or its unexpected termination which could result in an overall cataclysmic shutdown i.e. failure of the system on one machine should not affect other machines on the network. Also by establishing a self-configuration model the COBMAF framework should be able to take proper measures in cleaning up buffers, restarting agent communication and network links as well as make sure that the databases of the LocalAMS and Directory Facilitators reflect the most recent state of the agents.

The COBMAF Solution: To make the framework failsafe and self configurable, a number of procedures were implemented. Firstly, in the event of an interruption / unexpected termination in agent communication, exception messages were introduced which give the agent co-ordinator sufficient time in understanding the nature of failure and take appropriate measures such as shutting down the agent, removing its reference in the databases, restarting the agent, etc. Secondly, to account for the failure over network connections, exception messages were introduced such as “AMS connection Error” which clearly gives the agent co-ordinator sufficient information that the link between a particular agent and AMS / DF was not successfully established. Thirdly, the failsafe properties of the Visibroker Smart Agent were exploited.

Test Scenario & Results: To solve the problem of failsafe / self-configuration within the COBMAF framework, the system was run on independent machines on a dedicated network hosting heterogeneous software agents. The Local AMS and DF were started and then the RMA agent was launched. However there seemed to be a problem with this agent’s interface and an “Unexpected Runtime Error, Please restart

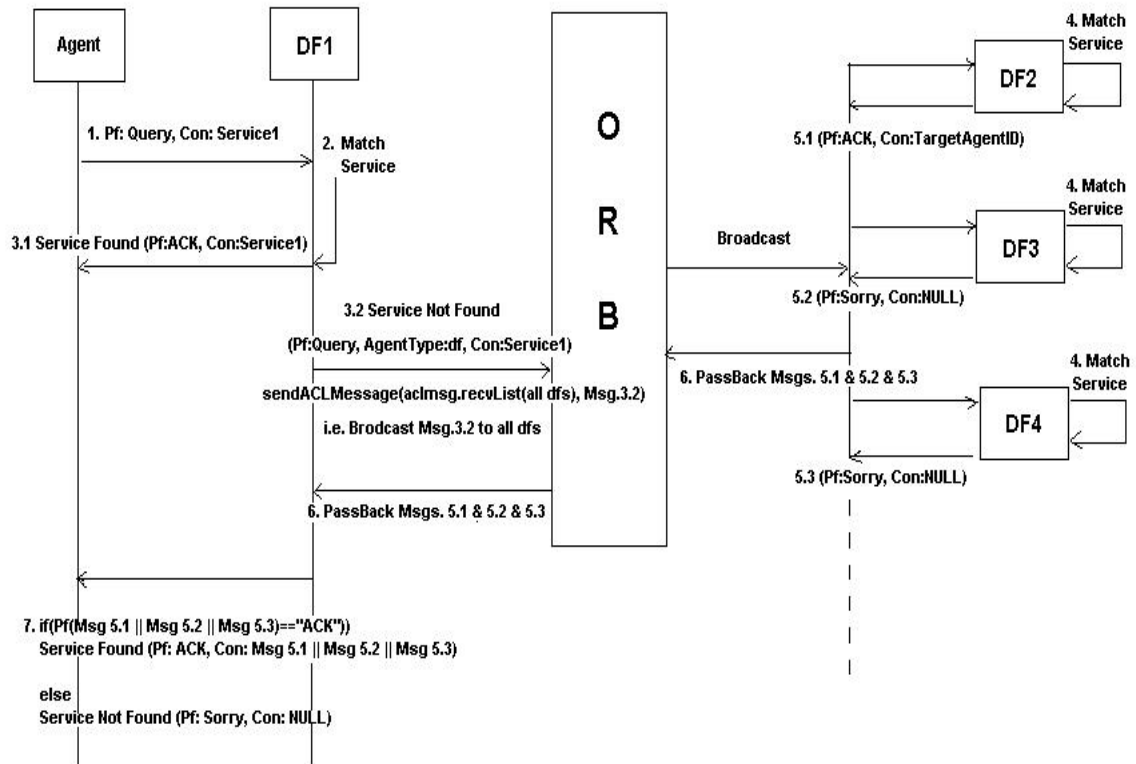
Agent ...” message was thrown by the system. The agent co-ordinator thereby got sufficient time to restart the agent and references with the databases were automatically cleaned up. It was found that the network did not get affected due to such unexpected agent interface failure and was able to successfully recover itself thereby accounting for its self-configuration nature. The COBMAF team also concluded that if a heavy services lookup load is necessary, it is advisable to use the VisiBroker Naming Service since it provides persistent storage capability and cluster load balancing whereas the Smart Agent only provides a simple round robin on a per `osagent` basis. On the other hand, due to the in-memory design of the Smart Agent, even if it is terminated by a proper shutdown or an abnormal termination, it does not failover to another Smart Agent in the same ORB domain i.e. to the same `OSAGENT_PORT` number, whereas the VisiBroker Naming Service provides such failover functionality. Hence it is solely left on the discretion of the agent co-ordinator whether to use either of the two.

9.4 Negotiation strategy (DF-DF)

Definition: The negotiation strategy is an algorithm or a series of communication events through the Agent Communication Language (ACL) between facilitates understanding and information interchange between two or more DFs by using the DF-Ontology found within the COBMAF framework.

Purpose: To help the agents registered with one DF to locate the services offered by other agents who may belong to some other DF either on a different machine or network for the purpose of either soliciting those services immediately or referencing them for future contextual usages.

The COBMAF Solution: To solve the problem of DF-DF negotiation strategy within the COBMAF framework, the following Agent-DF-DF-Agent communication model was adopted (*Extract from section 4.7*)



The search function is a bit more complicated since this requires that the complete cycle of Agent-DF-DF-Agent be completed. The search for a particular service on the Local DF1 is exactly similar to the query phase of Modify service as detailed in section 4.6.1. However, the next phase is different. Instead of directly sending a Sorry Message, DF1 now sends a Query message on the ORB indicating the Service name and the agent type as df. This broadcast message is sent by utilizing the `addreceiver()` function of `ACLMessage` class and thereby generating a receiver list containing all agents of type df. The message is then sent on the ORB and reaches each of the other DF agents i.e. DF2, DF3, DF4, etc. which have already been registered on the ORB with ids df2, df3, df4, etc. When this message is received by each of these DFs a matching search is performed on the local database of these DFs. If the particular service is found then the ID of the target agent to which that service belongs is returned. Remember that the ID of all agents on the ORB is unique and hence by getting a reference to a particular agent's ID it can be directly referenced in the future by the caller Agent. Now if either of the messages returned by all these DFs is positive, then the Agent has obtained the service and ID of the agent on the network to which it belongs and the Agent can now reference the target agent. If all the

messages by these DFs are negative, then the Agent has to wait for some time and query again or query over IIOP to a different ORB i.e. different network.

Test Scenario: In order to successfully test the adoption of the DF-DF negotiation strategy, the following test scenario was developed and tested:

The system was run over a dedicated network hosting heterogeneous software agents. Independent DFs were started on each of the machines and agents were launched which immediately registered their services with their respective DFs. Machine A's agent X then initiated a message with performative 'QUERY' to search for its desired service 'RMA-Pull' on its host machine. Since X was unable to achieve success in finding its desired service, DF1 forwarded this query to the ORB which was sent as a broadcast message on the network. Individual DFs received this message and queried their respective directories to successfully return 'RMA-Pull' service to X with a reference to DF3's (target DF) rma3 agent. We were then able to initiate further communication between X and rma3.

9.5 Co-ordination / Synchronization Model

Definition: In multi-agent system, coordination is the process of systematically analyzing a situation, developing relevant information, and informing appropriate command authority (agent or agent host) of viable alternatives for selection of the most effective combination of available resources to meet specific objectives. The coordination process can either be intra-system i.e. within the same MAS or inter-system across MAS.

Purpose: To facilitate coordination through the process of effective ACL communication among agents and thereby reduce the risk of deadlocks on shared resources or execution of simultaneous similar operations which may emphasize over-demand of the network bandwidth or over-numbered not-responded-to type of communication messages which may queue the receiver's mailboxes endlessly.

The COBMAF Solution: To solve the problem of co-ordination within the COBMAF framework, all the behavior methods of the individual agents are tuned to

the capacity of being able to effectively invoke themselves at particular instances and intervals of time. So if another agent say Agent X is utilizing resource Y then Agent Z's behavior method will cause the agent to sleep for a while and again wake up after a particular pre-defined interval. It is left to the developer's sense of understanding of the coordination model and his discretion as to which values would best suit effective resource and communication coordination between the COBMAF agents.

Test Scenario: To solve the problem of coordination within the COBMAF framework, the system was run on independent machines on a dedicated network hosting heterogeneous software agents. The Local AMS and DF were started and then the RMA agent was launched. Now, while the RMA agent was busy registering itself with the LocalAMS and DF, another agent i.e. Scheduler agent was launched. Without the implementation of an effective co-ordination model, this registration of the Scheduler agent would have failed since the RMA was requesting resources of the LocalAMS and DF. However, this was not the case and the Scheduler agent's behavior method prompted it to sleep for a while and again wake up when the RMA agent was done with its registration. The Scheduler agent was thus able to successfully register itself with the LocalAMS and DF.

9.6 Naming Service / Discovery Support

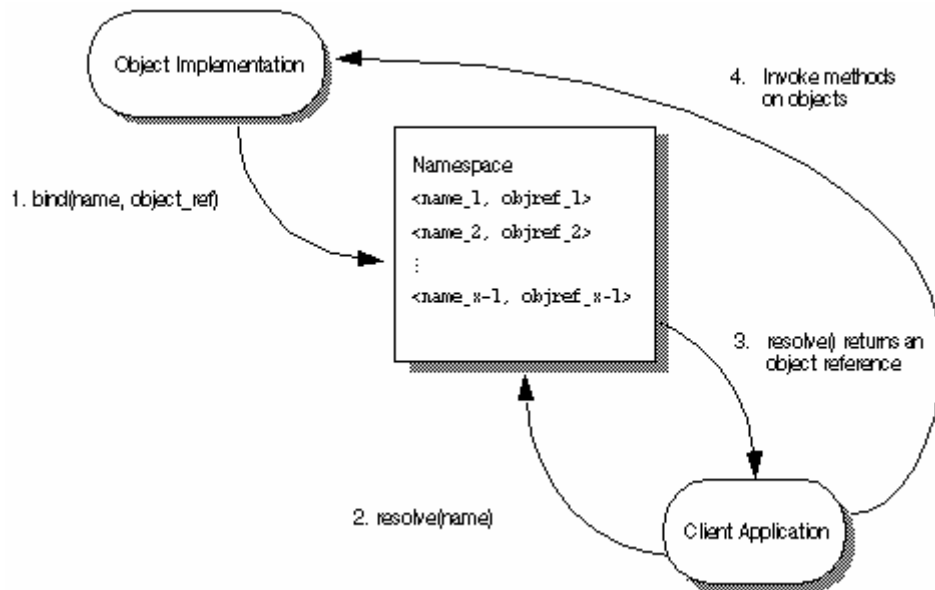
Definition: The Naming Service / Discovery Support is a mechanism to associate meaningful names to individual object implementations and extract them as required.

Purpose: To reduce the complexity of locating and retrieving objects from the thousands of objects available.

The COBMAF Solution: To solve the problem of naming service / discovery support within the COBMAF framework, we have an implementation of OMG's CORBA Naming Service specification called VisiBroker Naming Service which provides a CORBA 2.0-compliant solution ensuring flexible, heterogeneous interoperability. The naming service permits association of one or more *logical* names with an object implementation and store those names in a *namespace*.

The figure below contains a simplified view of the Naming Service that shows how

1. Object implementation *binds* a name to one of its objects within a *namespace*
2. Client applications can then use the same namespace to resolve a name and obtain an object reference



Secondly, the COBMAF framework has the Smart Agent which is a dynamic, distributed directory service that provides facilities used by both client programs and object implementations. When the client program (in this case, software agent) invokes `bind()` on an object, the Smart Agent is automatically consulted. The Smart Agent locates the specified implementation so that a connection can be established between the client and the implementation. When an object or implementation is deactivated, the Smart Agent removes it from the list of available objects.

Test Scenario & Results: In order to successfully test the adoption of a naming service / discovery support model for COBMAF, the system was run using Visibroker Naming Service & Smart Agent and the following observations were deduced:

- 1) The Smart Agent imposes no hard limits on the numbers and types of objects that it can support. Hence, there are reasonable best practices that can be followed when incorporating it into a larger architecture.

2) Since all objects' registered services are stored in memory, scalability cannot be optimized and be fault-tolerant at the same time.

3) Applications should use well known objects to bootstrap to other distributed services so as not to rely on the Smart Agent for all directory needs.

9.7 Interoperability and portability

9.7.1 Interoperability

Definition: The ability of multi-agent systems to operate in conjunction with each other encompassing different communication protocols, hardware software, application, and data compatibility layers

Purpose: In order for the applications to be truly autonomous and decentralized, heterogeneous multi-agent systems themselves must communicate and interoperate with each other. Multi-agent system interoperability allows heterogeneous agents to communicate across multiple systems and achieve designated goals, on a dynamic basis.

The COBMAF Solution: To solve the problem of interoperability within the COBMAF framework, we have taken a two-step approach. Firstly, underlying communication-level interoperability is achieved by the adoption of the CORBA framework which supports communication to take place between heterogeneous software components. In the case of COBMAF these components are nothing but the independent software agents. Secondly, higher level communication is achieved through the use of a single Agent Communication Language (ACL) which in this framework is nothing but a choice between 3 different ACL languages: FIPA-ACL, CSL and KQML. The developer thus has even got the freedom of implementing his choice of higher-level communication interoperability backed by COBMAF's proprietary ontology called COBONTO which serves as the 3rd tier of the interoperability model.

Test Scenario: In order to successfully test the adoption of an interoperability model for COBMAF, the following scenario was developed and tested.

Each of the 4 software agents, i.e. RMA, Scheduler, Procurement and Sales were developed in 2 programming languages namely Java and C++. Then these agents were run independently on 2 different machines and by using Visibroker ORB (Object Request Broker) bi-directional communication was successfully tested. This was still the primary interaction link and to test communication at higher levels, the COBMAF team dedicatedly coded a set of routines which were used by these agents to strike a semantically meaningful bi-directional dialog. Both the FIPA-ACL and COBMAF's proprietary CSL ACL were used at this higher interoperability level. To further support the context of communication, specialized ontologies such as Trade-Ontology were written which facilitated the meaning of the ensuing dialog between 2 agents. Thus, this and other similarly developed test scenarios show that interoperability within the COBMAF framework can be achieved at various levels irrespective of the language in which the agent is written or the machine on which it resides or even the network protocols.

9.7.2 Portability

Definition: Portability is the adaptation of a piece of software (in this case, software agents) so that it will function in a different computing environment to that for which it was originally written.

Purpose: The motivation behind porting COBMAF's source code is its capability to support any future releases of Visibroker from Inprise, Java Development Kit (JDK) & Swing GUI API from Sun Microsystems as well as virtually any other ORB such as JavaORB, OpenORB, etc.

The COBMAF Solution: Replacement of the block of code detailed in section 3.5.1 by code in section 3.5.2 to migrate from BOA to POA and thereby solve the problem of portability.

Test Scenarios: In order to successfully test the adoption of a portability model for COBMAF, the following scenarios were developed and tested.

The COBMAF code was run independently in 3 different modes:

- 1) BOA with Visibroker 3.0, JDK 1.1.5, AWT API
- 2) POA with Visibroker 6.5, JDK 1.5, Swing API
- 3) BOA / POA with OpenORB, JavaORB, JDK 1.1.5 / JDK 1.5, AWT / Swing API

It was found that the COBMAF code resulted in similar output performances with measurements on the lines of speed, memory, network usage and client requests.

10. Future Scope

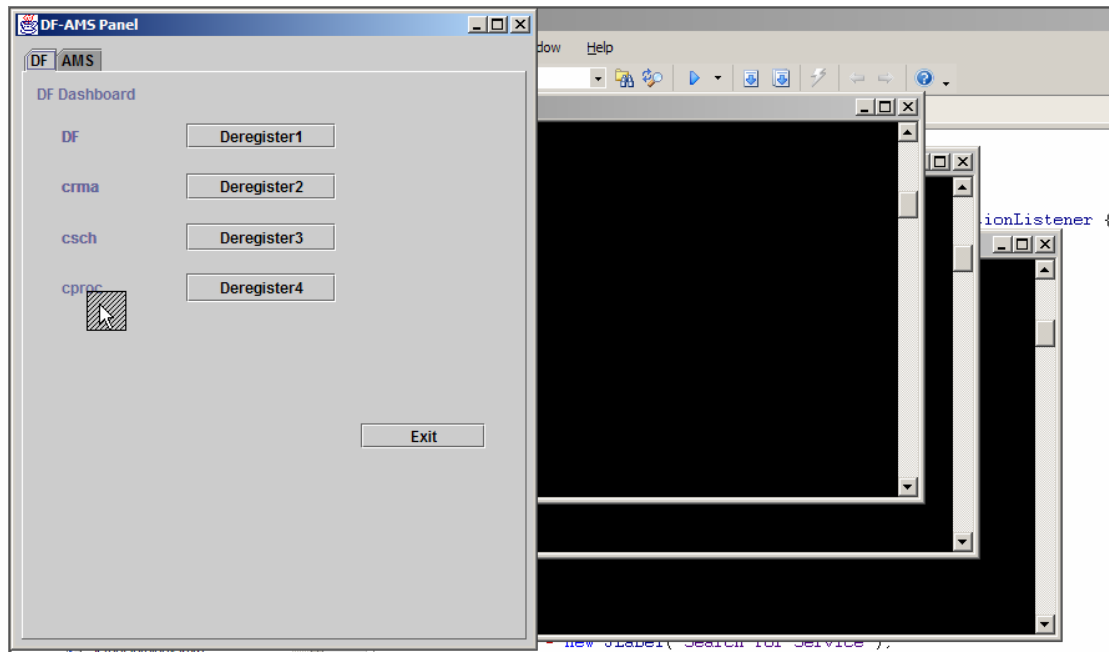
Our project has scope for further enhancements in the directions of extending the web model, incorporating security measures as well as further standardization for agent communication and greater support for content languages.

We propose that it can be extended in the following manner.

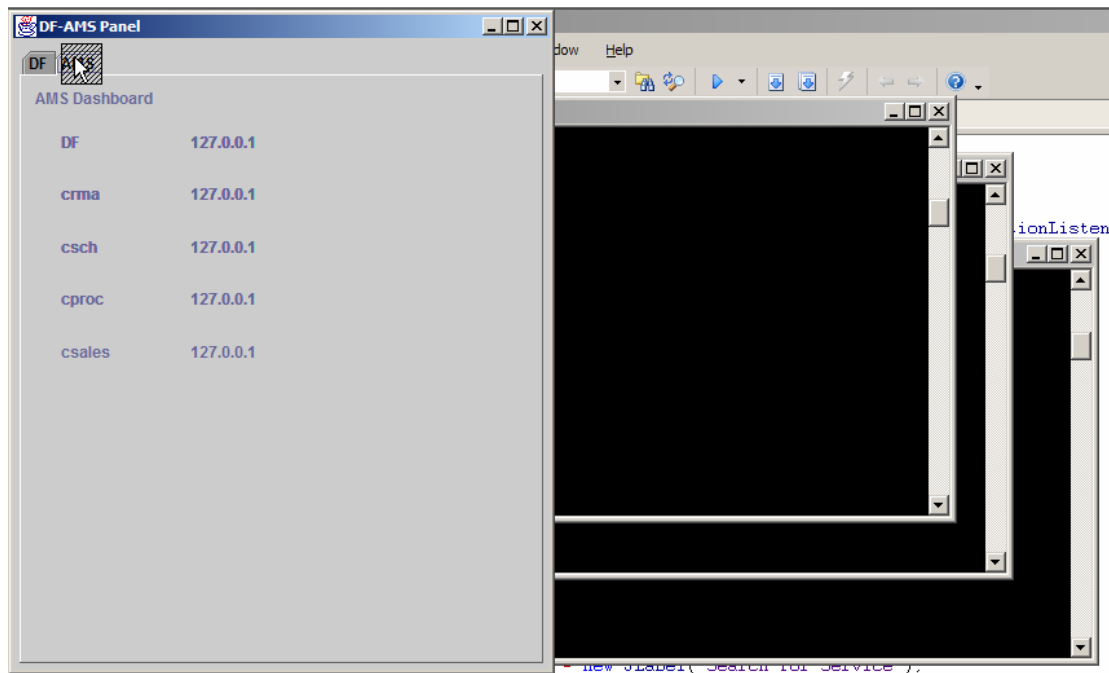
1. Incorporate a gatekeeper into COBMAF version 2.0 framework.
2. Extend the Web Model by usage of pushlets and call-back methods. Test the Web Model by requesting test-bed from AgentCities.net.
3. Extend the support for more number of ACL languages such as KQML and development of associated codecs.

We believe that the above 3 steps will increase the security & versatility of our agents and enable them to discover, interact with other agents not only across platforms on the same LAN but truly over the entire internet.

11. SCREENSHOTS:



COBMAF in action showing registered agents and way to de-register them.

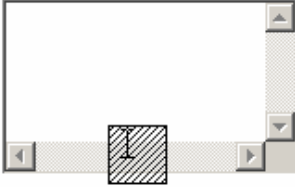


Agents Registered with Local AMS.

SCHEDULER FOR baleno

ID 1

NAME OF SPARE PART

DESCRIPTION ABOUT SPARE PART 

DURATION IN THE ASSEMBLY LINE

PRECEDING TASKS IN THE ASSEMBLY LINE 1 2

OK

Negotiation Process – Filling up the details.

```

c:\C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Content: cproc 127.0.0.1 procurement
Content Language --> CSL
Ontology --> DF-Ontology

crma
SEND:::from: c To: crma
crma
RECU:::from: c To: crma
Performative: UPDATE
Content: csales 127.0.0.1 sales

Content Language --> CSL
Ontology --> DF-Ontology

Content Language -> CSL
Ontology -> Trade-Ontology
crma 127.0.0.1
Encrypted performative is 5PURQ7YR
Encrypted content is onyRAB

Content Language --> CSL
Ontology --> Trade-Ontology

```

COBMAF Ontology

```

C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Ontology --> DF-Ontology

Service Count for Agent: csales 127.0.0.1:: 2
csales 127.0.0.1
Sales Push
Trade

*****
DF Agent Services Dashboard
*****
Agent Name: crma 127.0.0.1
Service Name: rma pull
Agent Name: csch 127.0.0.1
Service Name: Schedluer Push
Agent Name: cproc 127.0.0.1
Service Name: Procurement
Agent Name: cproc 127.0.0.1
Service Name: Trade
Agent Name: csales 127.0.0.1
Service Name: Sales Push
Agent Name: csales 127.0.0.1
Service Name: Trade

```

DF Agent Services Dashboard

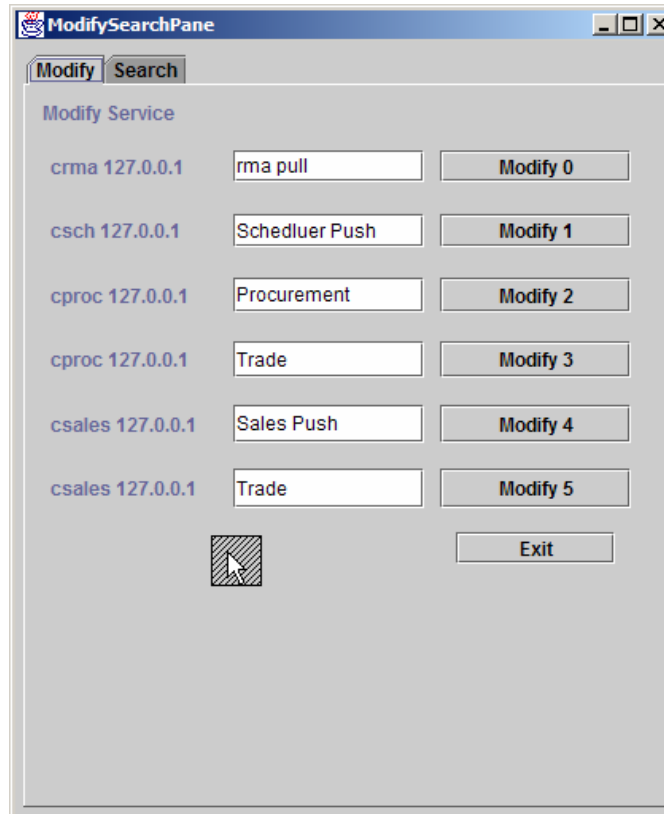
SCHEDULER FOR baleno

ID	NAME	DESCRIPTION	DURATION	PRECEDING TASKS	START	NEW START	END	NEW END	SLACK
1	tyre	good	1	tyre	1	1	2	2	0
2	lights	good	3	tyre	2	2	5	5	0

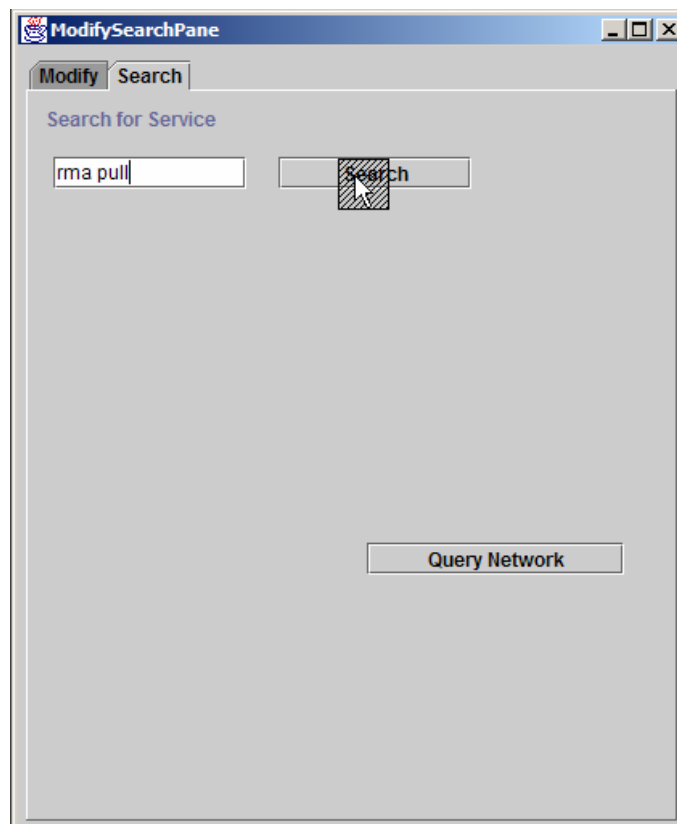
THE MAXIMUM TIME TAKEN TO ASSEMBLE THE CAR IS :

QUIT

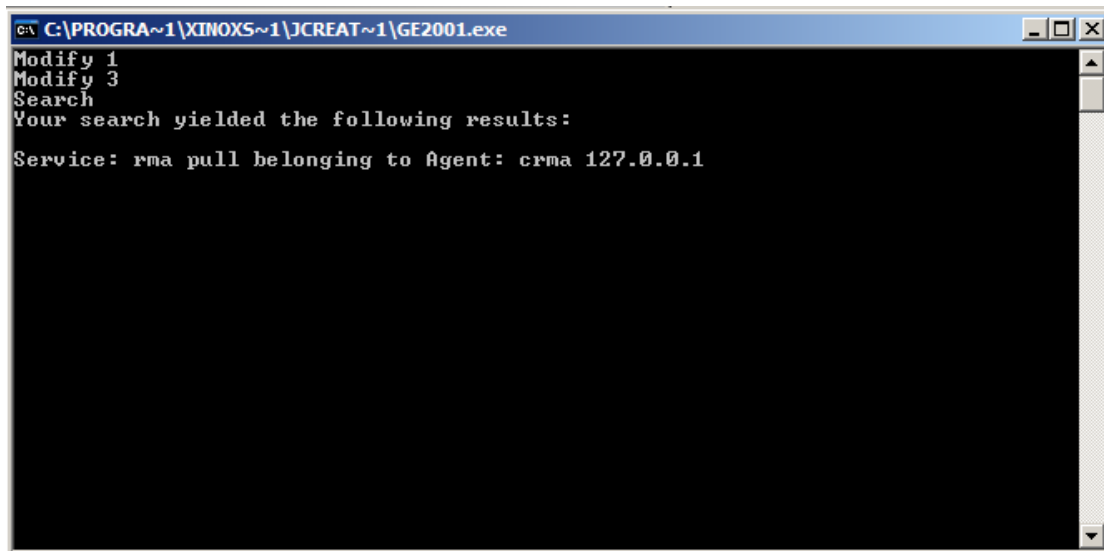
Negotiation Process.



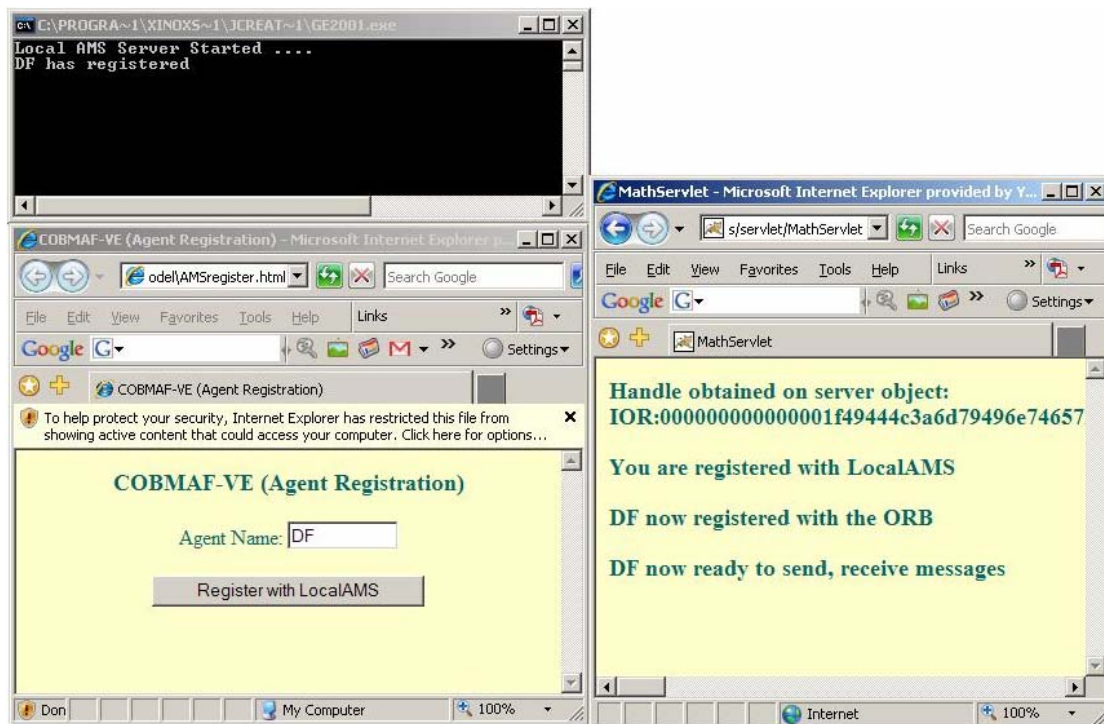
Modify Pane



Search Pane



Search Result.



COBMAF Web Model.