Project Report On

VIRTUAL TRAVEL AGENCY IN SEMANTIC WEB

Submitted in partial fulfillment of the requirements for
the degree of

BACHELOR OF ENGINEERING

In

COMPUTER ENGINEERING

Submitted by

R.Dhana Nandini	5801
Rege Mayuri Dilip	5840
Gaurav Sharma	5845

Under the guidance of

Professor Swati Ringe
(Assistant Professor, Computer Engineering Department)

Department of Computer Engineering
Fr. Conceicao Rodrigues College of Engineering
Bandra, Mumbai-400 050

CERTIFICATE

This is to certify that the following students working on the project "**VIRTUAL TRAVEL AGENCY IN SEMANTIC WEB**" have satisfactorily completed the requirements of the term work as partial fulfillment of the course B.E in Computer Engineering of the University of Mumbai during academic year 2012-2013 under the guidance of

Prof. Swati Ringe.

Submitted By: R.Dhana Nandini(5801)
Rege Mayuri Dilip (5840)
Gaurav Sharma (5845)

_____ _____

Internal Examiner **Internal Guide**

_____ _____

External Examiner **Head of the Department**

_____

Principal

Computer Engineering Department
Fr. Conceicao Rodrigues College of Engineering Bandra, Mumbai – 400 050

ACKNOWLEDGEMENT

We owe our sincere thanks to **Ms Swati Ringe** for her continuous support and guidance throughout the course of project .

TABLE OF CONTENTS

1.PREFACE

Imagine a “Virtual Traveling Agency”, called VTA for short, which is an end user service providing eTourism services to customers. These services can cover all kinds of information services concerned with tourism information – from information about events and sights in an area to services that support booking of flights, hotels, rental cars, etc. online. Such VTAs are already existent, currently those portals are accessible via html sites. The partners of the VTA are integrated via conventional B2B integration.

By applying Semantic Web Services, a VTA will be more easily be able to configure an invoke Web Services provided by several eTourism suppliers and aggregate them into new customer services. Such VTAs providing automated eTourism services to end users via different interfaces and can be more easily reconfigured according to the actual needs.

2.ABSTRACT

The project focuses mainly on Semantic Web Services and Semantic Web.

The Semantic Web is a vision of a Web which can be interpreted by computer programs. The purpose of Semantic Web is sharing and reusing knowledge. Users will be able to search more accurately of the information and the services they need from the tools provided. Semantic Web focus on metadata, ontologies , logic and inference, and software agents.

Semantic Web Services(SWS) add semantics to the web service allowing interoperability as well as reasoning to create a comprehensive response adapted to user goals.Using SWS,we can add higher semantic levels to the existing frameworks, to improve their usage and ease scalability.

Virtual Travel Agency is an intermediary between customer and tourism service providers.It provides tourism services to customers by aggregating services from various tourism service providers.

Web services in our project are flight ticket booking and hotel booking.We offer services to end user who sends a trip booking request to VTA.

3.INTRODUCTION

3.1 AIMS AND OBJECTIVES

Our VTA aggregates Web Services of different tourism service providers. In a nutshell it provides the following functionality: A customer uses the VTA service as the entry point for his requests. These end-user services are aggregated by the VTA by invoking and combining Web Services offered by several tourism service providers. To facilitate this, there can be a so called "umbrella" contract between the service providers and the VTA for regulating usage and allowance of the Web Services.

Thus in short,it will fulfill two main objectives:

- To reduce the human efforts in deciding the tourism service by letting the machine provide maximum service.

- To provide related services to the user depending upon its requirement.

3.2 PROBLEM STATEMENT

A Web Service of a Virtual Travel Agency, short: VTA, offers end-user services for searching and booking hotel and flight tickets.
This Web Service is composed out of other Web Services, namely one for searching existing flight and hotel connections, and one for booking flight and hotel tickets online.

The flow of the project shall be the following:

- The customer specifies his high level goal of purchasing hotel and flight tickets.
- The VTA is one of the Service Providers that is identified as suitable fulfilling his request
- The customer specifies his requirements specification (time, date, rating, location,etc).
- The VTA returns a set of possible connections.
- The user selects one of these connections and poses a request for booking the ticket online
- The VTA books the ticket for the customer and gives him the confirmation.

3.3 SCOPE

The scenario outlines a general structure for VTAs that can be extended to more complex scenarios wherein the customer can be a Web Service itself, thus creating a network of composed services that offer complex tourism services.

For example,
1. One VTA can provide flight booking services for an airline union,
2. Second VTA aggregates booking service for a worldwide hotel chain, and
3. Third VTA provides booking services for rental cars by combining the services of several worldwide operating car rental agencies.
4. Then, a Fourth VTA uses these services for providing an end-user service for booking complete holiday trips worldwide.

General tasks under the scope of the VTA are where the inputs to VTA will be

Flight → source, destination, date, time, price

Hotel → city, fromdate , todate , price, no of people

1.Search Flights for the particular location
2. Book Flight from source to destination.
3. Search Hotels for the particular location
4. Book Hotel for the specified location

4.LITERATURE SURVEY

4.1.SEMANTIC WEB

Web 3.0 is a Read/Write/Execute Web. The data is present in such a manner that it is not only human readable but also machine processable[11].The **problem** which the world was facing before the Web 3.0 was as follows-

Whenever users wanted to search anything on the internet say a user wants to see the shoes within the range of 2000-5000 She enters "SHOES WITHIN 2000-5000RS" in the text box of a search engine. On a traditional Web, the search engine will display the links of all those pages in which only the keywords entered in the query will be present. From there, the user has to explore every link and see whether the information is related to his context.

DISADVANTAGE :

It can really be an irritation to explore every link and also waste of MB's if the information is not relevant.

The **solution** to the above problem is web 3.0.The approach taken by the latest web is that it will search the pages where the information is present relevant to the user's context and directly display a complete list of all the shoes along with their exact prices according to the user's wish means that it doesn't display the links but gives direct output.[1]

All this is achieved through web 3.0's core software technology of **artificial intelligence** which can intelligently learn and understand the semantics.Thus making application of Web 3.0 more personalised and accurate and intelligent.

CHARACTERISTICS OF WEB 3.0

- **Intelligence :**

  Using various artificial intelligence techniques like fuzzy sets, neural networks, machine learning etc,applications can work intelligently with the use of Human-Computer interaction.

- **Personalisation :**

  Individual preferences will be considered while performing different activities such as information processing,search etc.

- **Interoperability :**

  It deals with reuse of information that facilitates exchange of new knowledge.

- **Virtualisation :** It will create high end 3D graphics that will be utilised in various graphic applications.

4.2.WEB SERVICE:

Web services are a new breed of Web application. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions, which can be anything from simple requests to complicated business processes

Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service.

Modular: Components are reusable and it is possible to compose them into larger components[2].

Available: Services must be exposed outside of the particular paradigm or system they are available in. Business services can be completely decentralized and distributed over the Internet. The dynamic enterprise and dynamic value chains become achievable and may be even mandatory[2].

Described: Services have a machine-readable description that can be used to identify the interface of the service[2].

Implementation-independent: The service interface is independent of the ultimate implementation.

Published: Service descriptions are made available in a repository where users can find the service and use the description to access the service.

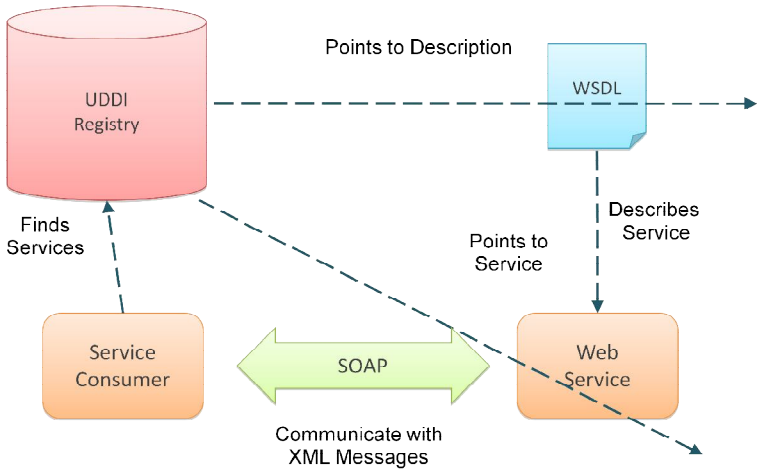


Fig #1 Web Services

4.3.SEMANTIC WEB SERVICES

With the development of the semantic web, Semantic Web services (SWS) became an important role.
Semantic web services expand the capabilities of a web service by associating a semantic description of the web service in order to enable automatic search, discovery,selection, composition, and integration across heterogeneous users and domains[1].

Current technologies allow usage of Web Services but it provides only syntactical information descriptions.It also provides:

- syntactic support for discovery, composition and execution
- Web Service usability, usage, and integration needs to be inspected manually
- no semantically marked up content / services
- no support for the Semantic Web
- current Web Service Technology Stack failed to realize the promise of Web Services

Fig #2 Semantic Web Services

5.TECHNOLOGIES TO IMPLEMENT SEMANTIC WEB SERVICES

5.1 Web Service Modelling Ontology(WSMO)

The *Web Service Modeling Ontology* (WSMO) provides a conceptual framework for semantically describing Web services and their special properties. WSMO can be applied for semantic web service discovery[1][3].

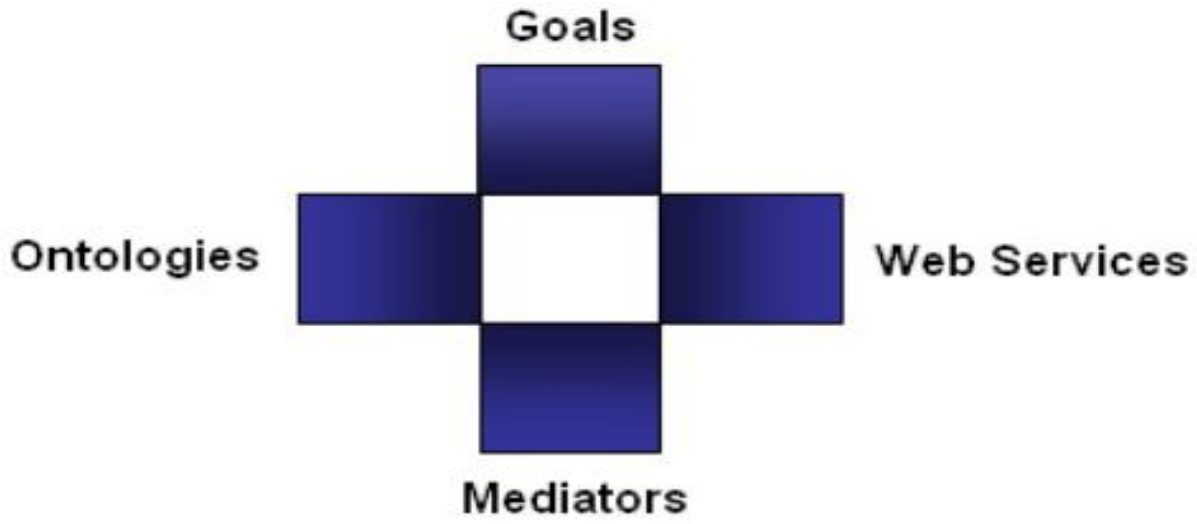


Fig #3 WSMO Elements

5.2 OWL-S

OWL-S is an ontology built of top of the Web Ontology Language (OWL) for describing semantic Web services. The OWLS ontology has three main parts (sub-ontologies): the service profile, the process model and the service grounding[3].

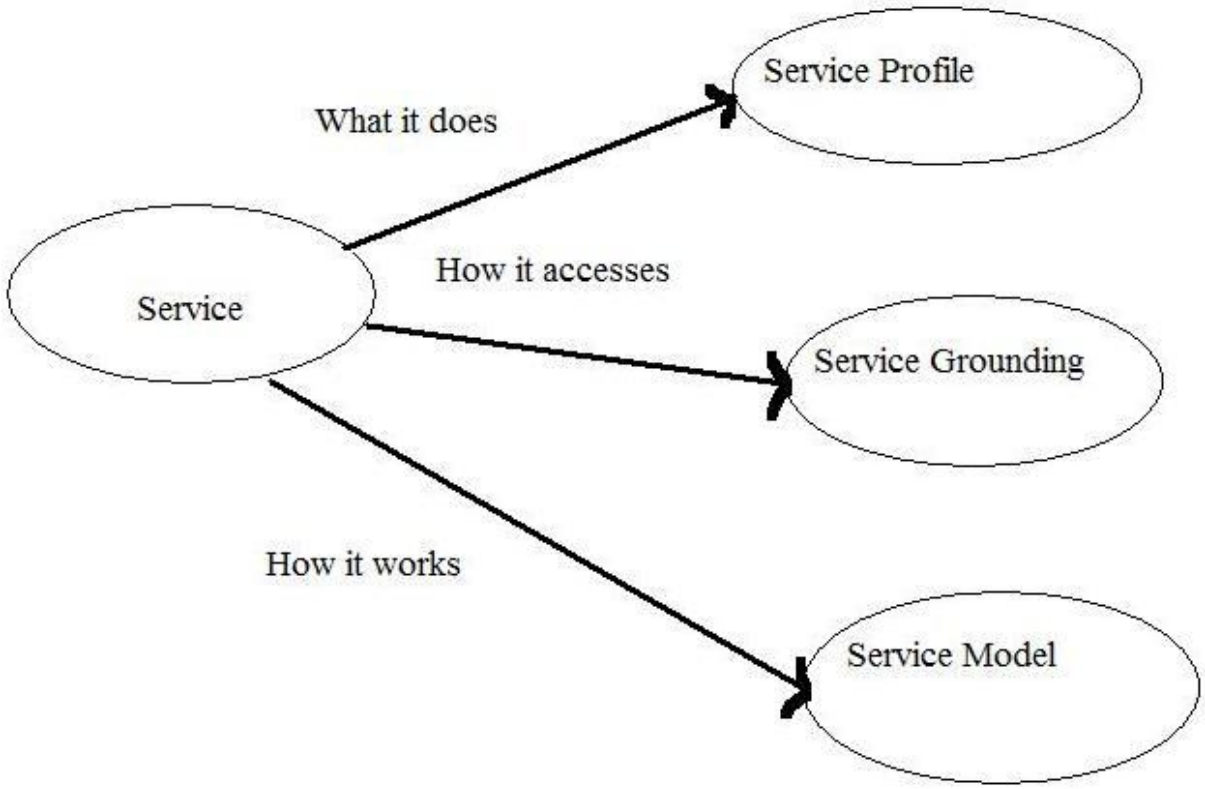


Fig #4 OWL- Services

The service profile describes what the service does, in a way suitable for automatic service discovery.

The service profile includes the service name, provider information, the inputs and outputs of the service, the preconditions required by the service, a description of what the service accomplishes, and the expected effects that result from execution of the service. The profile may also provide service classification information, as well as information about the quality of service it provides.

The service grounding specifies the details of how to access a service. Typically, the grounding will specify a communication protocol, message formats, and other service-specific details, such as port numbers used in contacting the service.

The OWL-S submission describes the complementary relationship between the service grounding and the WSDL.

The service model describes how a service works, including how to invoke, compose and monitor the service. This description includes the sets of inputs, outputs, pre-conditions and results of the service execution. The service model can be viewed as a process - a specification of the ways a client may interact with a service.

6.EXISTING SYSTEMS

Problems :

- **Some websites have centralized database systems while some have distributed databases.There is no reliability in centralized systems while distributed systems can be difficult to implement.**

- **Tedious and Time consuming task**
  A typical (Business-to-Consumer) B2C scenario involves using online websites for such as shopping and travelling. The consumer's interest lies in choosing the best deal from available deals. Best could be quantified by the customer as a function of factors such as price, reputation of manufacturer and seller, quality of service of the seller and convenience.
  Traditionally, there have been two approaches: (1) visit each website individually and interact with it, or (2) use a service that interacts with a fixed number of websites and aggregate the data for the user.
  Such services (websites) help the user compare similar services offered by multiple B2C sites. The former approach is a tedious and time-consuming task and involves a high cognitive load for the user. The latter, although it is easier for the user, typically relies on hard-coded logic embedded in wrappers that interact with the B2C sites. As a result, integrating a new B2C site is a substantial engineering task and such systems need to be updated whenever changes are made to the websites that are used as information sources.

- **Searches for the same information in different digital libraries**
  Example: I want travel from Innsbruck to Rome.

Information may come from different web sites and needs to be combined

- Example: I want to travel from Innsbruck to Rome where I want to stay in a hotel and visit the city.

- **Automatic Discovery of web services is not provided**

- **Automatic Invocation of Web Services not provided**
  Machine-readable descriptions of existing web service applications are stored in the form of WSDL files. The WSDL file essentially provides information about the operations (methods) that can be invoked, defines the schema for the message content, describes the message exchange pattern and the location of the service. This is analogous to a collection of methods and their signatures along with information about the syntax of the input and the output. The decision to invoke is still done manually, based on textual descriptions that explain the implications of invoking the service.

- **Dynamic Binding (Choreography) of Web Services not provided**

- **Mediation of data not done**

All the above problems are handled in VTA.

Our approach is as follows

We have used the following technologies

1.Java Agents using JADE

2.Ontologies

These technologies realize the semantic web services.

The goal of SWS is to make the existing account of services richer by articulating the implications in SWL rather than text. The SWS lets a machine know when a service should/could be executed and what the effects of executing the service are.

This description can, in principle, be used by the **agent** to perform **automatic discovery, automatic composition and automatic invocation**.

A web-service-based system with rich service descriptions coupled with a user agent offers solutions to some of these problems. The client provides flexibility by relieving the customer of the tedious task of navigating through the site; at the same time, it has the potential to interact with an arbitrary number of web-based businesses, as long as it has the service descriptions for those web sites. Since the client relies solely on the service description to interact with the service, the service description needs to be updated every time the service provider decides to change some part of the service.

7.PROPOSED SYSTEM

Technologies used:

7.1. Ontology

Ontologies are the structural frameworks for organizing information on the semantic Web and within semantic enterprises. ontologies often have a linked or networked "graph" structure.Multiple things can be related to other things, all in a potentially multi-way series of relationships. A distinguishing characteristic of ontologies compared to conventional hierarchical structures is their degree of connectedness, their ability to model coherent, linked relationships.

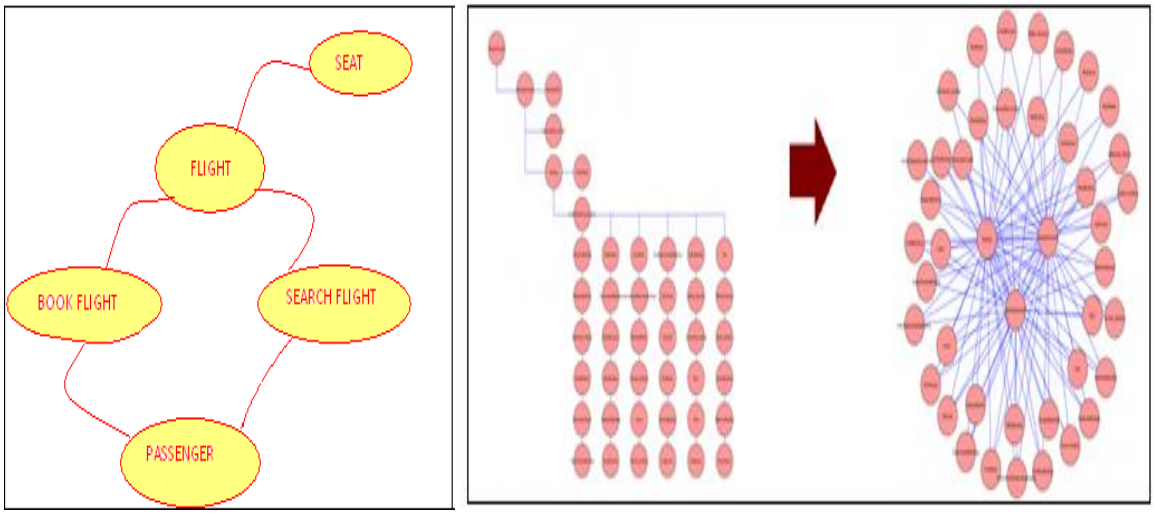


Fig #5 . A sample ontology and an ontology showing how its different from conventional hierarchical structures.

Ontologies in VTA

HotelOntology :

Code # 1

```
package solarforce.bean;
import jade.content.onto.*;
import jade.content.schema.*;

@SuppressWarnings("serial")
public class HotelOntology extends Ontology implements HotelVocabulary {
// The singleton instance of this ontology
public static Ontology theInstance = new HotelOntology();
// This is the method to access the singleton music shop ontology object
public static Ontology getInstance() {
return theInstance;
```

```
}
// Private constructor
private HotelOntology() {

// The music shop ontology extends the basic ontology
super(ONTOLOGY_NAME, BasicOntology.getInstance());
try {
add(new ConceptSchema(PERSON), Person.class);
add(new ConceptSchema(HOTEL), Hotel.class);
add(new ConceptSchema(ROOM), Room.class);
add(new PredicateSchema(ROOM_ALLOTED), Room_Alloted.class);
add(new AgentActionSchema(BOOK_ROOM), BookRoom.class);
// Structure of the schema for the Item concept

ConceptSchema cs = (ConceptSchema) getSchema(HOTEL);
cs.add(HOTEL_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING),
ObjectSchema.MANDATORY); // The serial-number slot is optional and
// allowed values are integers.
// Structure of the schema for the CD concept
cs.add(STREET,                                          (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(CITY,                                            (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(RATING,                                          (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(OFFERS,          (PrimitiveSchema)          getSchema(BasicOntology.STRING),
        ObjectSchema.OPTIONAL);
cs.add(NO_OF_ROOMS_AVAILABLE,                           (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs = (ConceptSchema) getSchema(ROOM);
cs.addSuperSchema((ConceptSchema) getSchema(HOTEL));
cs.add(ROOM_NO,                                         (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(ROOM_TYPE,                                       (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY); // The tracks slot has
cardinality > 1
// Structure of the schema for the Track concept
cs.add(FARE,                                            (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(CHECK_IN,                                        (PrimitiveSchema)
getSchema(BasicOntology.DATE),ObjectSchema.OPTIONAL);
cs.add(CHECK_OUT,                                       (PrimitiveSchema)
getSchema(BasicOntology.DATE),ObjectSchema.OPTIONAL);
cs.add(CAPACITY,                                        (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);


cs = (ConceptSchema) getSchema(PERSON);
```

```
cs.add(PERSON_NAME,(PrimitiveSchema)getSchema(BasicOntology.STRING),ObjectSch
ema.MANDATORY);
cs.add(AGE,                                                    (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(SSN,                                                    (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(GENDER,                                                 (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(ADDRESS,                                                (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(CREDIT_CARD,                                            (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(PASSWORD,                                               (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);


// Structure of the schema for the Owns predicate
PredicateSchema ps = (PredicateSchema) getSchema(ROOM_ALLOTED);
ps.add(ROOM_ALLOTED_PERSON, (ConceptSchema) getSchema(PERSON));
ps.add(ROOM_ALLOTED_ROOM, (ConceptSchema) getSchema(ROOM));
// Structure of the schema for the Sell agent action

AgentActionSchema as = (AgentActionSchema) getSchema(BOOK_ROOM);
as.add(BOOK_ROOM_PERSON, (ConceptSchema) getSchema(PERSON));
as.add(BOOK_ROOM_ROOM, (ConceptSchema) getSchema(ROOM));
}
catch (OntologyException oe) {
oe.printStackTrace();
}
}
}
```

Hotel Vocabulary:

Code # 2

```
package solarforce.bean;
public interface HotelVocabulary {

  //-------> Basic vocabulary
  public static final int NEW_HOTEL = 1;
  public static final int BOOK_A_ROOM = 2;
  public static final int SEARCH = 3;
  public static final int UPDATE = 4;


 //error codes


  public static final int NO_SUCH_HOTEL = 20;
```

```
    public static final int ROOM_NOT_AVAILABLE = 21;
    public static final int ILLEGAL_OPERATION = 22;

    //ontology vocabulary

     public static final String ONTOLOGY_NAME = "Hotel-ontology";

     public static final String ROOM_ALLOTED = "Room_Alloted";
     public static final String ROOM_ALLOTED_ROOM = "Room";
     public static final String ROOM_ALLOTED_PERSON = "Person";


     public static final String BOOK_ROOM = "BookRoom";
     public static final String BOOK_ROOM_ROOM = "Room";
     public static final String BOOK_ROOM_PERSON = "Person";

     public static final String HOTEL = "Hotel";
     public static final String HOTEL_NAME= "hname";
     public static final String STREET = "street";
     public static final String CITY = "city";
     public static final String RATING = "rating";
     public static final String OFFERS = "offers";
     public static final String NO_OF_ROOMS_AVAILABLE= "tavailable";


     public static final String PERSON = "Person";
     public static final String PERSON_NAME = "name";
     public static final String GENDER="gender";
     public static final String AGE = "age";
     public static final String SSN = "ssn";
     public static final String ADDRESS = "address";
     public static final String CREDIT_CARD = "credit_card";
     public static final String PASSWORD= "password";


     public static final String ROOM = "Room";
     public static final String ROOM_NO = "room_no";
     public static final String ROOM_TYPE = "rtype";
     public static final String FARE = "fare";
     public static final String CHECK_IN = "check_in";
     public static final String CHECK_OUT = "check_out";
     public static final String CAPACITY = "capacity";
}
```

Hotel Ontology Graphical View:

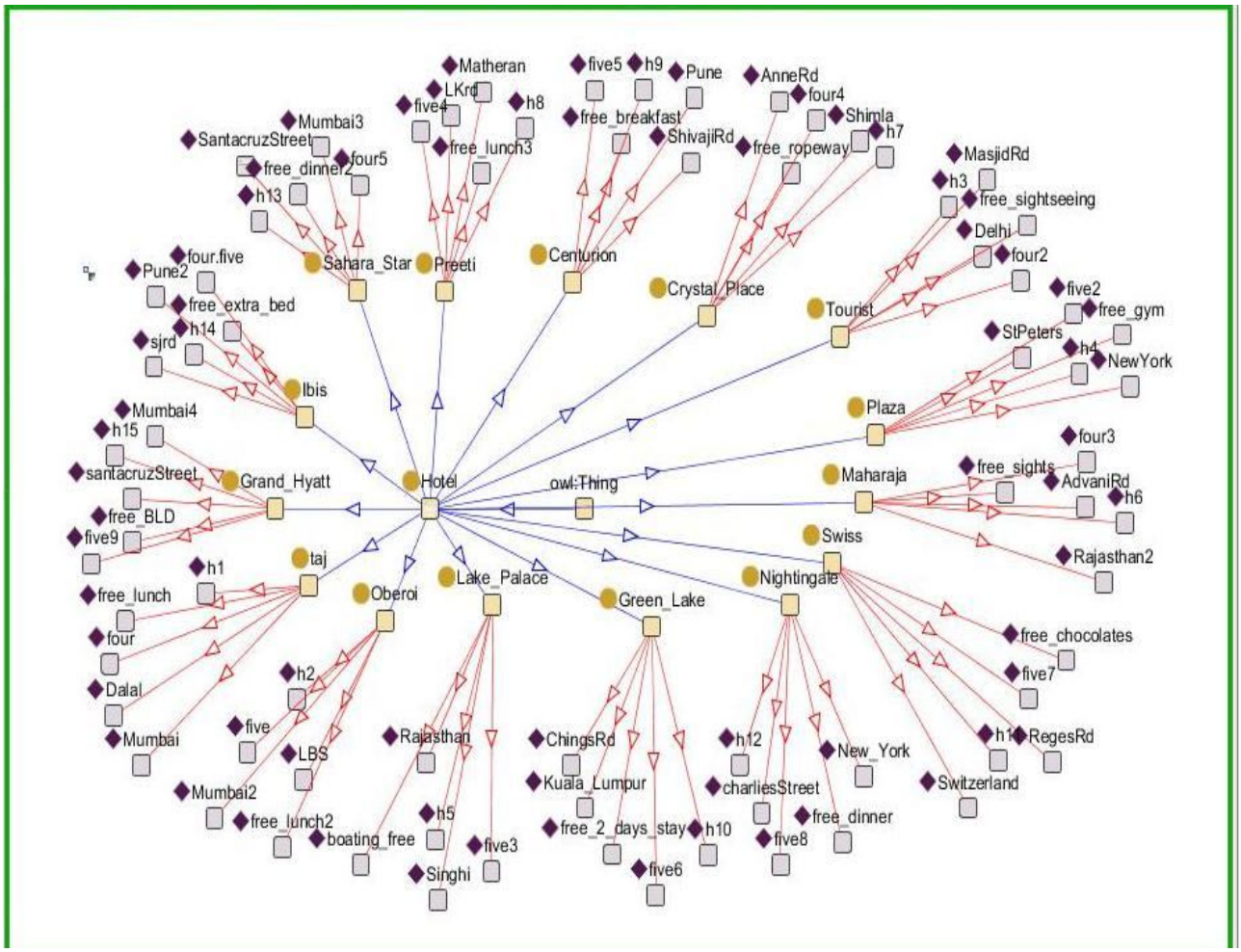


Fig #6 Hotel Ontology

Flight:

Flight Ontology:
Code #3

```
import jade.content.onto.*;
import jade.content.schema.*;

public class FlightOntology extends Ontology implements FlightVocabulary {
// The singleton instance of this ontology
private static Ontology theInstance = new FlightOntology();
// This is the method to access the singleton music shop ontology object
public static Ontology getInstance() {
return theInstance;
}
// Private constructor
private FlightOntology() {
```

```
// The music shop ontology extends the basic ontology
super(ONTOLOGY_NAME, BasicOntology.getInstance());
try {
add(new ConceptSchema(PASSENGER), Passenger.class);
add(new ConceptSchema(FLIGHT), Flight.class);
add(new ConceptSchema(SEAT), Seat.class);
add(new PredicateSchema(SEAT_ALLOTED),Seat_Alloted.class);
add(new AgentActionSchema(BOOK_FLIGHT), Book_Flight.class);
add(new AgentActionSchema(SEARCH_FLIGHT), Search_Flight.class);
// Structure of the schema for the Item concept

ConceptSchema cs = (ConceptSchema) getSchema(FLIGHT);
cs.add(FLIGHT_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING),
ObjectSchema.MANDATORY); // The serial-number slot is optional and
// allowed values are integers.
// Structure of the schema for the CD concept
cs.add(FNAME,                                          (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(FNO,                                            (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(FTYPE,                                          (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(TICTYPE,                                        (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(TICFARE,                                        (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(RETDAY,                                         (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(RETMONTH,                                       (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(RETYEAR,                                        (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(DEPDAY,                                         (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(DEPMONTH,                                       (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(DEPYEAR,                                        (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(RETIME,                                         (PrimitiveSchema)
getSchema(BasicOntology.DATE),ObjectSchema.MANDATORY);
cs.add(DEPTIME,                                        (PrimitiveSchema)
getSchema(BasicOntology.DATE),ObjectSchema.MANDATORY);
cs.add(LEAVEFROM,                                      (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(GOINGTO,                                        (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
```

```
cs.add(SEAT_NO,                                          (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(SAVAILABLE,                                       (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);


cs.add(PASSENGER,                                        (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(PASSENGER_NAME,(PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
s.add(PASSENGER_EMAIL,                                   (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(PASSENGER_PHONE,                                  (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.MANDATORY);
cs.add(ADDRESSTREET,                                     (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.OPTIONAL);
cs.add(ADDRESSCITY,                                      (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(ADDRESSTATE,                                      (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.OPTIONAL);
cs.add(ADDRESSCOUNTRY,                                   (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(ADDRESSPINCODE,                                   (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),ObjectSchema.OPTIONAL);

cs.add(CREDIT_CARD_NO,                                   (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(CREDIT_CARD_TYPE,                                 (PrimitiveSchema)
getSchema(BasicOntology.STRING),ObjectSchema.MANDATORY);
cs.add(CREDIT_CARD_EXPIRATION,                           (PrimitiveSchema)
getSchema(BasicOntology.DATE),ObjectSchema.MANDATORY);

// Structure of the schema for the Owns predicate
PredicateSchema ps = (PredicateSchema) getSchema(SEAT_ALLOTED);
ps.add(SEAT_ALLOTED_PASSENGER, (ConceptSchema) getSchema(PASSENGER);
ps.add(SEAT_ALLOTED_SEAT, (ConceptSchema) getSchema(SEAT));
// Structure of the schema for the Sell agent action

AgentActionSchema as = (AgentActionSchema) getSchema(BOOK_FLIGHT);
as.add(BOOK_FLIGHT_PASSENGER, (ConceptSchema) getSchema(PASSENGER));
as.add(BOOK_FLIGHT_FLIGHT, (ConceptSchema) getSchema(FLIGHT));

AgentActionSchema as = (AgentActionSchema) getSchema(SEARCH_FLIGHT);
as.add(SEARCH_FLIGHT_NAME, (ConceptSchema) getSchema(FLIGHT));
}
catch (OntologyException oe) {
oe.printStackTrace();
}
}
```

```
}
```

Flight Vocabulary:
Code # 4

```
package ontologies;

public interface HotelVocabulary {

  //-------> Basic vocabulary
  public static final int NEW_FLIGHT = 1;
  public static final int BOOK_FLIGHT = 2;
  public static final int SEARCH_FLIGHT = 3;
  public static final int UPDATE_SEAT = 4;


 //error codes

  public static final int SEAT_NOT_AVAILABLE = 21;
  public static final int ILLEGAL_OPERATION = 22;

  //ontology vocabulary
        public static final String ONTOLOGY_NAME = "FlightOntology";

        public static final String SEAT_ALLOTED = "Seat_Alloted";

        public static final String SEAT_ALLOTED_PASSENGER = "Passenger";
        public static final String SEAT_ALLOTED_SEAT = "Seat";


        public static final String BOOK_FLIGHT = "Book_Flight";
        public static final String BOOK_FLIGHT_FLIGHT = "Flight";
        public static final String BOOK_FLIGHT_PASSENGER = "Passenger";

        public static final String SEARCH_FLIGHT = "Search_Flight";
        public static final String SEARCH_FLIGHT_NAME = "Flight";

   public static final String FLIGHT = "flight";
   public static final String FLIGHT_NAME= "fname";
   public static final String FLIGHT_NO = "fno";
   public static final String FLIGHT_TYPE = "ftype";
   public static final String TICKET_TYPE = "tictype";
   public static final String TICKET_FARE = "ticfare";
   public static final String RETURN_DAY= "retday";
   public static final String RETURN_MONTH= "retmonth";
   public static final String RETURN_YEAR= "retyear";
   public static final String RETURN_TIME= "retime";
   public static final String DEPARTURE_DAY = "depday";
   public static final String DEPARTURE_MONTH = "depmonth";
```

```
public static final String DEPARTURE_YEAR = "depyear";
public static final String DEPARTURE_TIME = "deptime";
public static final String LEAVING_FROM = "leavefrom";
public static final String GOING_TO = "goingto";

public static final String SEAT = "Seat";
public static final String SEAT_NO = "seat_no";
public static final String NO_OF_SEATS_AVAILABLE= "savailable";


public static final String ADDRESS_STREET = "addresstreet";
public static final String ADDRESS_CITY = "addresscity";
public static final String ADDRESS_STATE = "addresstate";
public static final String ADDRESS_COUNTRY = "addresscountry";
public static final String ADDRESS_PINCODE = "addresspincode";
public static final String CREDIT_CARD_NO = "credit_card_no";
public static final String CREDIT_CARD_TYPE = "credit_card_type";
public static final String CREDIT_CARD_EXPIRATION = "credit_card_expiration";
public static final String PASSENGER = "passenger";
public static final String PASSENGER_NAME = "passenger_name";
public static final String PASSENGER_EMAIL = "passenger_email";
public static final String PASSENGER_PHONE = "passenger_phone";
```

Flight Ontology Graphical View:

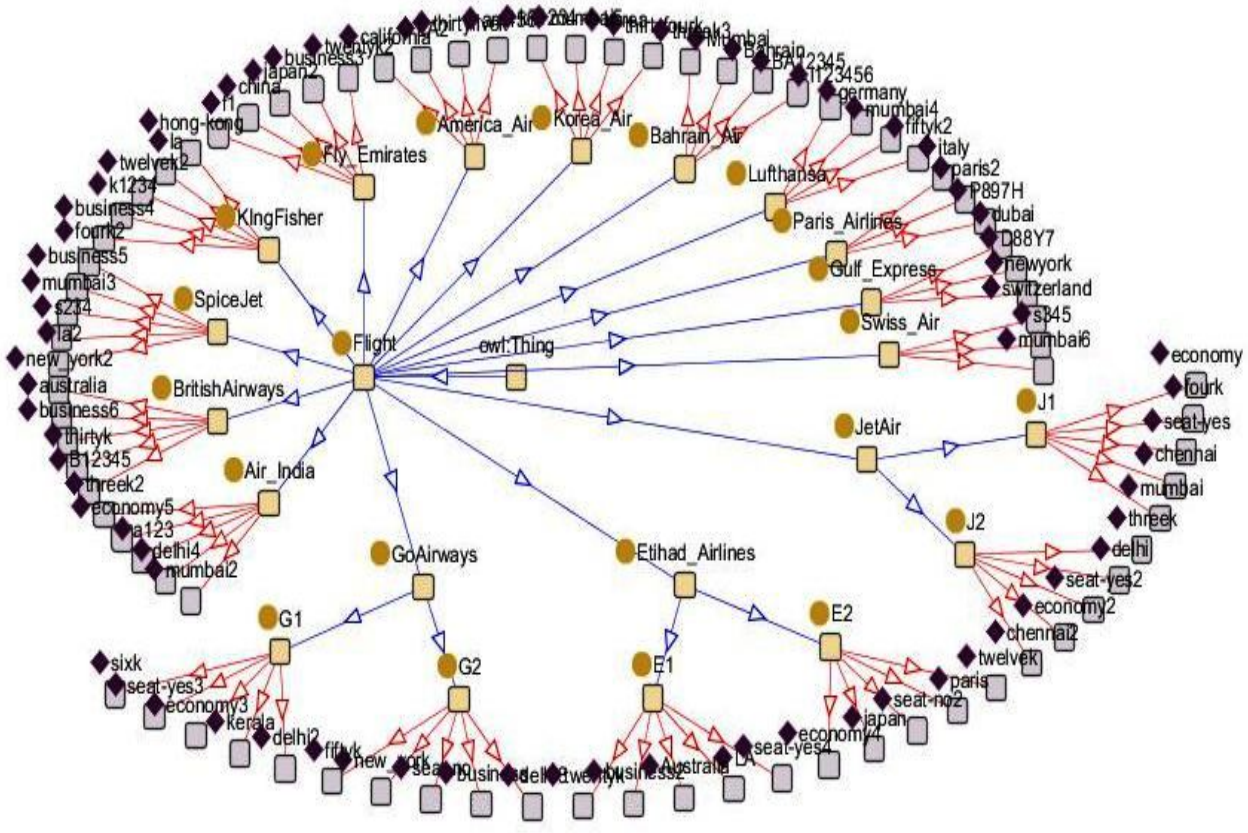


Fig #7 Flight Ontology

7.2.Multi-Agent Framework using JADE

A software agent is a computer program that acts for a user or other program in a relationship of agency. The attributes of a software agent that makes important from a SWS implementation point of view are that agents

- are not strictly invoked for a task, but activate themselves.
- do not require interaction of user.
- may invoke other tasks including communication.

In short Agents are independent entities. A Multi-Agent System (MAS) consists of a team or organization of software agents, collectively performing a task which could not be performed by any individual agent.

JADE is a middleware that facilitates the development of multi-agent systems. It includes

• A **runtime environment** where JADE agents can "live" and that must be active on a given host before one or more agents can be executed on that host.
• A **library** of classes that programmers have to/can use (directly or by specializing them) to develop their agents.
• A suite of **graphical tools** that allows administrating and monitoring the activity of running agents.

Each running instance of the JADE runtime environment is called a ***Container*** as it can contain several agents. The set of active containers is called a ***Platform***.
A single special ***Main container*** must always be active in a platform and all other containers register with it as soon as they start. It follows that the first container to start in a platform must be a main container while all other containers must be "normal" (i.e. non-main) containers and must "be told" where to find (host and port) their main container (i.e. the main container to register with).
If another main container is started somewhere in the network it constitutes a different platform to which new normal containers can possibly register. Figure 1 illustrates the above concepts through a sample scenario showing two JADE platforms composed of 3 and 1 containers respectively.
JADE agents are identified by a unique name and, provided they know each other's name, they can communicate transparently regardless of their actual location: same container (e.g. agents A2 and A3 in Figure), different containers in the same platform (e.g. A1 and A2) or different platforms (e.g. A4 and A5).

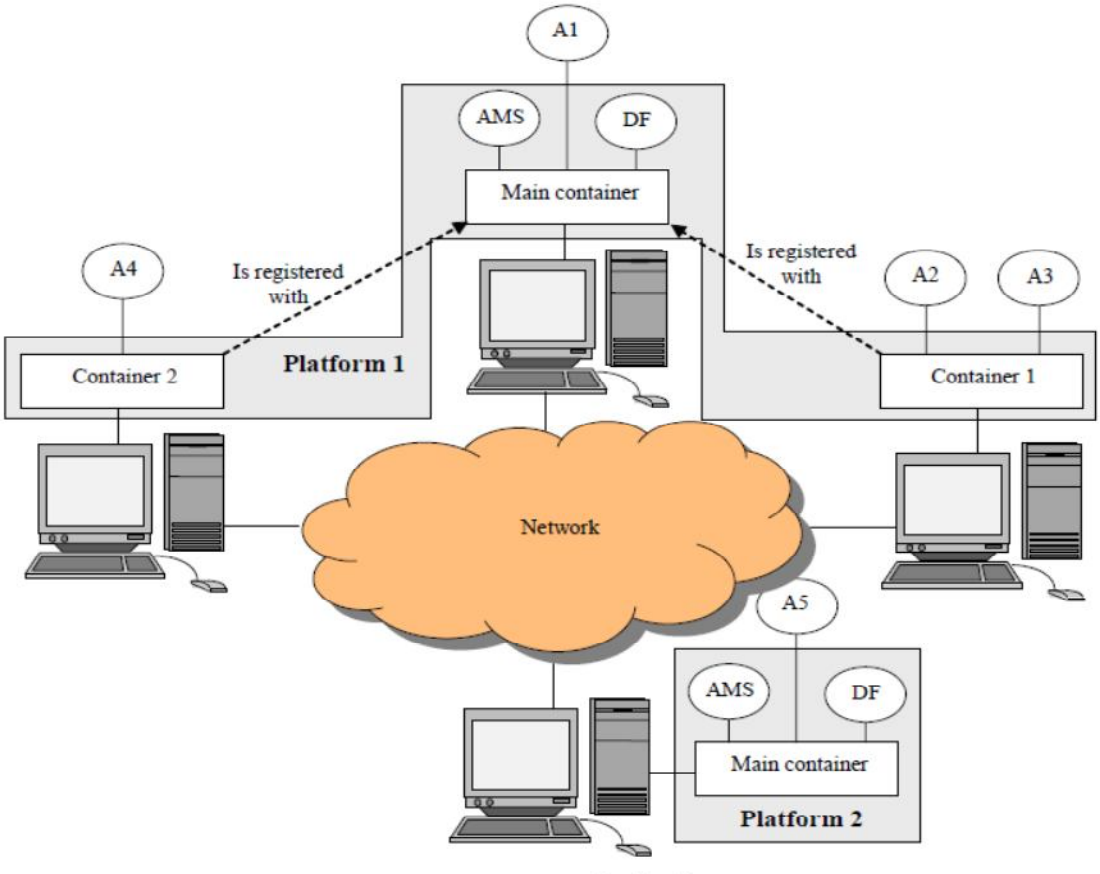


Fig #8 Containers and Platforms

The following command line launches a Main Container activating the JADE management GUI (-gui) option. <classpath> must include all jade classes plus all required application-specific classes.

java -cp <classpath> jade.Boot -gui

The following command line launches a peripheral container (-container option) that registers to a main container running on host avalon.tilab.com (-host option) and activates an agent called john of class myPackage.MyClass (-agents) option

java -cp <classpath> jade.Boot -container -host avalon.tilab.com -agents john:myPackage.myClass

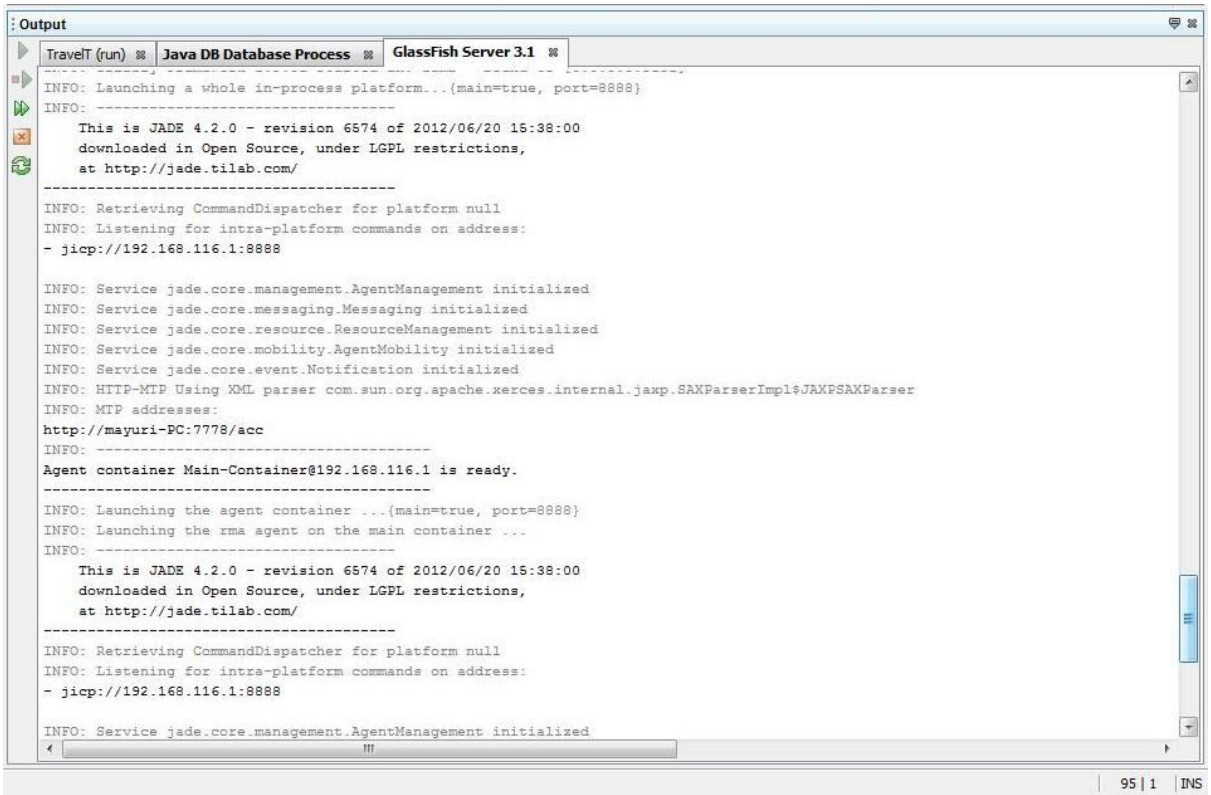


Fig # 9 Jade Output

AMS and DF

Besides the ability of accepting registrations from other containers, a main container differs from normal containers as it holds two special agents (automatically started when the main container is launched).
The **AMS** (Agent Management System) that provides the naming service (i.e. ensures that each agent in the platform has a unique name) and represents the authority in the platform

(for instance it is possible to create/kill agents on remote containers by requesting that to the AMS). This tutorial does not illustrate how to interact with the AMS as this is part of the advanced JADE programming.

The **DF** (Directory Facilitator) that provides a Yellow Pages service by means of which an agent can find other agents providing the services he requires in order to achieve his goals.

Creating Agents

Creating a JADE agent is as simple as defining a class extending the jade.core.Agent class and implementing the setup() method as shown in the code below.

Code #5

```
import jade.core.Agent;
public class BookBuyerAgent extends Agent {
protected void setup() {
// Printout a welcome message
System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
}
}
```

The setup() method is intended to include agent initializations. The actual job an agent has to do is typically carried out within "behaviours".

Agent Identifier

Each agent is identified by an "agent identifier" represented as an instance of the jade.core.AID class. The getAID() method of the Agent class allows retrieving the agent identifier. An AID object includes a globally unique name plus a number of addresses. The name in JADE has the form <*nickname*>@<*platform-name*> so that an agent called *Peter* living on a platform called *P1* will have ***Peter@P1*** as globally unique name. The addresses included in the AID are the addresses of the platform the agent lives in. These addresses are only used when an agent needs to communicate with another agent living on a different platform.

The agents in VTA are

1. Registration Agent RegPerson@192.168.116.1:8888/JADE

2. Login Agent Login@192.168.116.1:8888/JADE

3. Hotel Service 1 hs1@192.168.116.1:8888/JADE

4.Hotel Service 2 hs2@192.168.116.1:8888/JADE

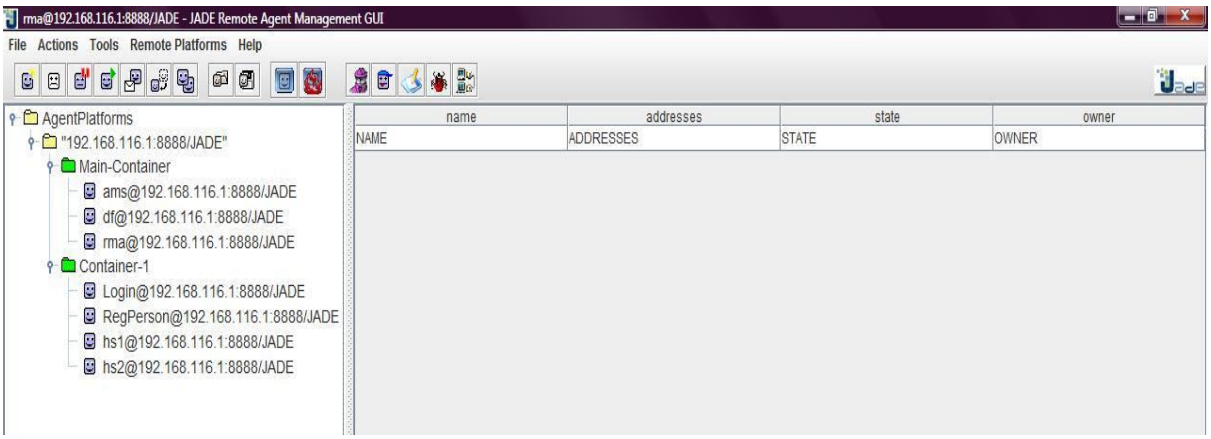


Fig # 10 Jade Agents in VTA

This is VTA Register Server Agent
Code #6

```
public class RegPersonServerAgent extends Agent {
   private ContentManager manager  = (ContentManager) getContentManager();
   // This agent "speaks" the SL language
   private Codec      codec    = new SLCodec();
   // This agent "knows" the Hotel ontology
private Ontology   ontology = HotelOntology.getInstance();

   @Override
protected void setup()
        {
        System.out.println("inside regperson server");

        manager.registerLanguage(codec);
        manager.registerOntology(ontology);
      addBehaviour(new HandleInformBehaviour(this));

                // pong behaviour

        }
//Cyclic Behaviour
class HandleInformBehaviour extends CyclicBehaviour {
        public HandleInformBehaviour(Agent a) {
                super(a);
        }
      @Override
public void action()
{
        ACLMessage                                msg                                =
receive(MessageTemplate.MatchPerformative(ACLMessage.INFORM));
```

```
        if(msg!=null)
        {
        System.out.println("upadte received from client");
        try
        {
                Object ce = msg.getContentObject();
        if(ce instanceof Person)
                {
                        Person p=(Person)ce;
                        System.out.println(p);
                RegisterClient r=new RegisterClient();
                String result=r.insertPerson_XML(p);
                r.close();
                if(result.equalsIgnoreCase("true"))
                {
                   System.out.println("registeration successful !!!!!");
                   ACLMessage reply = msg.createReply();
                                        reply.setPerformative( ACLMessage.INFORM );
                                        reply.setContent("true");
                                        send(reply);

                }
                else
                   System.out.println("registration failed");
                }
        }
        catch(Exception e)
        {
                e.printStackTrace();
        }
        }
}

}

   @Override
         protected void takeDown()
           {
                try { DFService.deregister(this); }
                catch (Exception e) {}
           }
}
```

Agent Behaviour

The actual job an agent has to do is typically carried out within “behaviours”.

A behaviour represents a task that an agent can carry out and is implemented as an object of a class that extends **jade.core.behaviours.Behaviour.** In order to make an agent execute the task implemented by a behaviour object it is sufficient to add the behaviour to the agent by means of the addBehaviour() method of the Agent class. Behaviours can be added at any time: when an agent starts (in the setup() method) or from within other behaviours.
Each class extending Behaviour must implement the action() method, that actually defines the operations to be performed when the behaviour is in execution and the done() method (returns a Boolean value), that specifies whether or not a behaviour has completed and have to be removed from the pool of behaviours an agent is carrying out.

Various Agent Behaviours

1.One-shot behaviours

.
"One-shot" behaviours that complete immediately and whose action() method is executed only once.
The jade.core.behaviours.OneShotBehaviour already implements the done() method by returning true and can be conveniently extended to implement one-shot behaviours.

Code #7

```
public class MyOneShotBehaviour extends OneShotBehaviour {
public void action() {
// perform operation X
}
}
```

Operation X is performed only once.

2.Cyclic Behaviour

"Cyclic" behaviours that never complete and whose action() method executes the same operations each time it is called. The jade.core.behaviours.CyclicBehaviour already implements the done() method by returning false and can be conveniently extended to implement cyclic behaviours.

Code #8

```
public class MyCyclicBehaviour extends CyclicBehaviour {
public void action() {
// perform operation Y
}
}
```

Operation Y is performed repetitively forever (until the agent carrying out the above behaviour terminates).

3. Generic Behaviours

Generic behaviours that embeds a status and execute different operations depending on that status. They complete when a given condition is met.

Code #9

```
public class MyThreeStepBehaviour extends Behaviour {
```

```
private int step = 0;
public void action() {
switch (step) {
case 0:
// perform operation X
step++;
break;
case 1:
// perform operation Y
step++;
break;
case 2:
// perform operation Z
step++;
break;
}
}
public boolean done() {
return step == 3;
}
}
```

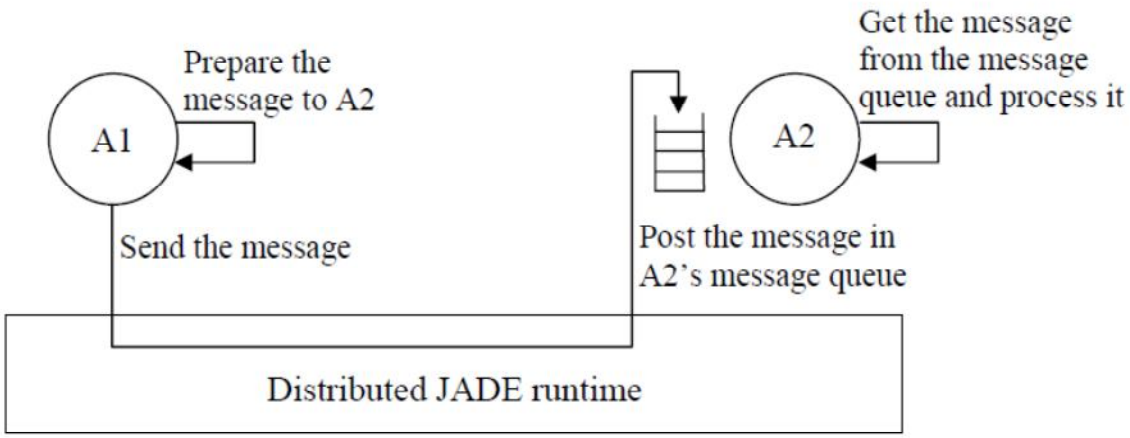
Operations X, Y and Z are performed one after the other and then the behaviour completes.

Behaviours required for VTA Agents

VTA uses **Cyclic** behavior for the agents(Ref. Code #2)

Agent Communication -ACL Messages

One of the most important features that JADE agents provide is the ability to communicate. The communication paradigm adopted is the **asynchronous message passing**. Each agent has a sort of mailbox (the agent message queue) where the JADE runtime posts messages sent by other agents. Whenever a message is posted in the message queue the receiving agent is notified.



Fig#11 Jade Asynchronous Message Passing

ACL Language

Messages exchanged by JADE agents have a format specified by the ACL language defined by the FIPA (http://www.fipa.org) international standard for agent interoperability. This format comprises a number of fields and in particular:

• The ***sender*** of the message
• The list of ***receivers***
• The communicative intention (also called "***performative***") indicating what the sender intends to achieve by sending the message.

• The ***content*** i.e. the actual information included in the message (i.e. the action to be performed in a REQUEST message, the fact that the sender wants to disclose in an INFORM message …)
• The content ***language*** i.e. the syntax used to express the content (both the sender and the receiver must be able to encode/parse expressions compliant to this syntax for the communication to be effective).

• The ***ontology*** i.e. the vocabulary of the symbols used in the content and their meaning (both the sender and the receiver must ascribe the same meaning to symbols for the communication to be effective).

A message in JADE is implemented as an object of the jade.lang.acl.ACLMessage class that provides get and set methods for handling all fields of a message.

Code #10

```
public void action()
{
        ACLMessagemsg=
receive(MessageTemplate.MatchPerformative(ACLMessage.INFORM));
        if(msg!=null)
        {
        System.out.println("upadte received from client in login server");
        try
        {
                Object ce = msg.getContentObject();
        if(ce instanceof Person)
                {
                        Person p=(Person)ce;
                        System.out.println(p);
                RegisterClient r=new RegisterClient();
                String result=r.login_XML(p);
                r.close();
                if(!result.equalsIgnoreCase("false"))
                   System.out.println("registeration successful !!!!!");
                else
                   System.out.println("registration failed");
                ACLMessage reply = msg.createReply();
```

```
                                        reply.setPerformative( ACLMessage.INFORM );
                                        reply.setContent(result);
                                        send(reply);

            }
    }
```

7.3.Multi-Agents with Ontologies:

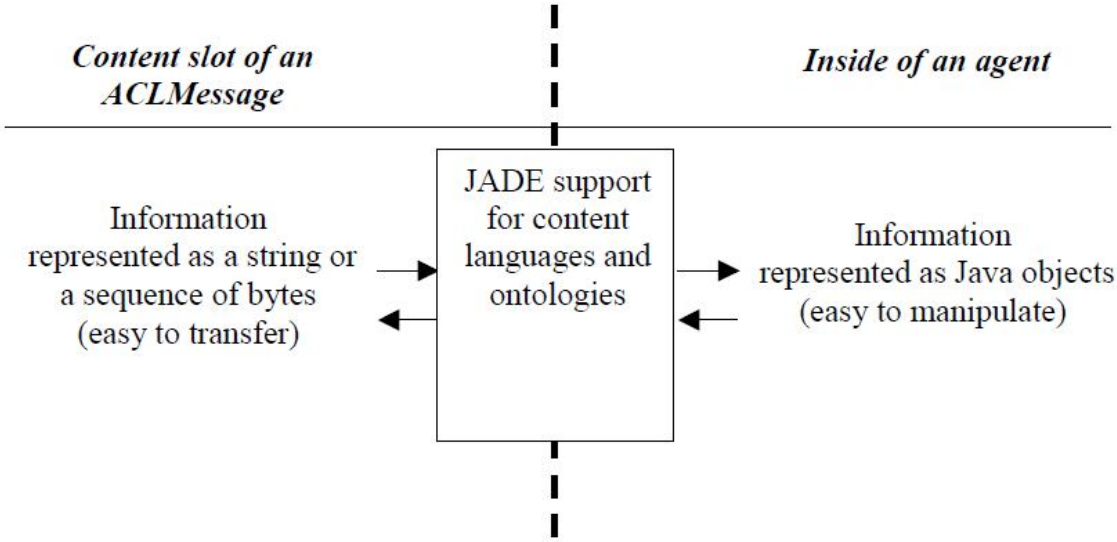


Fig #12 Conversion performed by JADE

Support for Ontology is provided in the jade.content package.
Ontology here is organized using three Schemas

1)Predicate Schema
2)Concept Schema
3)AgentAction Schema

Predicate Schema:
Predicates (or facts) are expressions that say something about the status of the world and can be true or false e.g.
(Works-for (Person :name John) (Company :name TILAB)) stating that "the person John works for the company TILAB".

Concept Schema
Concepts i.e. expressions that indicate entities with a complex structure that can be defined in terms of slots
e.g.(Person :name John :age 33),Room, Hotel, Flight

Agent Action Schema

Agent actions i.e. special concepts that indicate actions that can be performed by some agents
e.g. Registration, Login, Search,Book

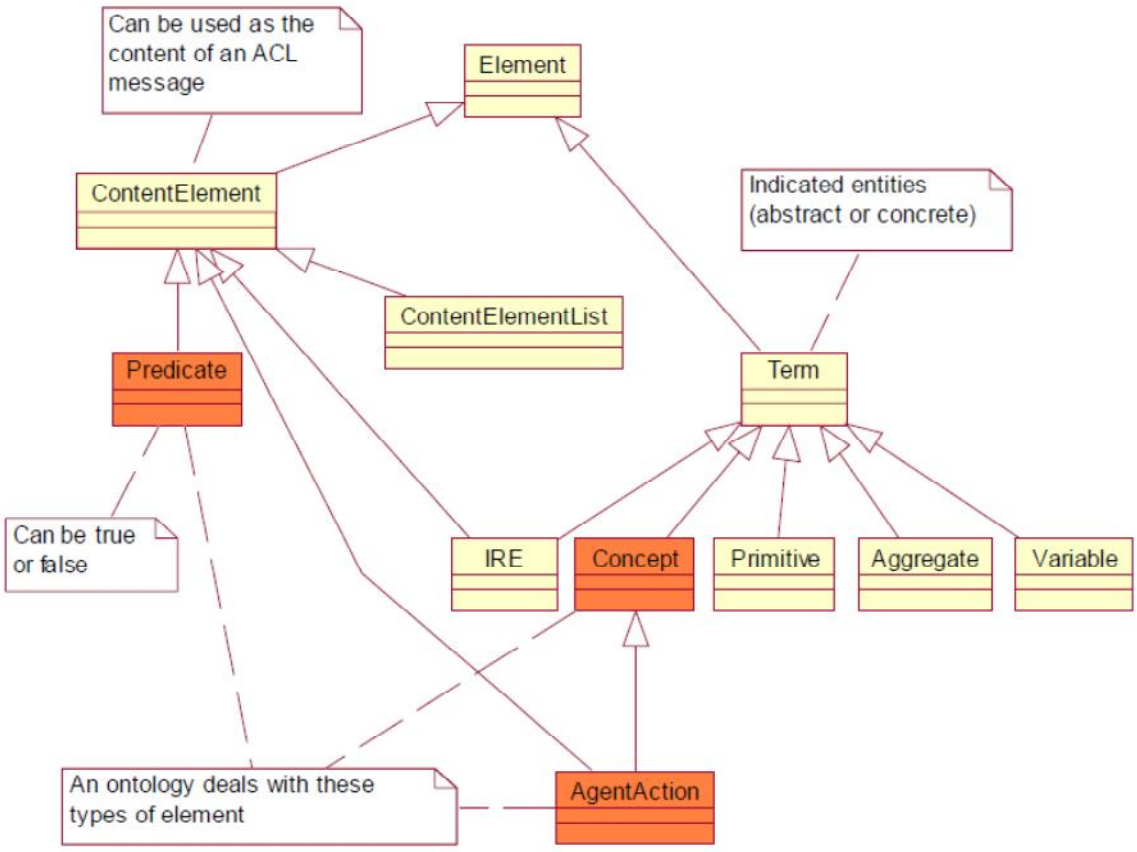


Fig # 13 The Agents Class Diagram

8.PROJECT DESIGN

8.1.Use Case Diagram

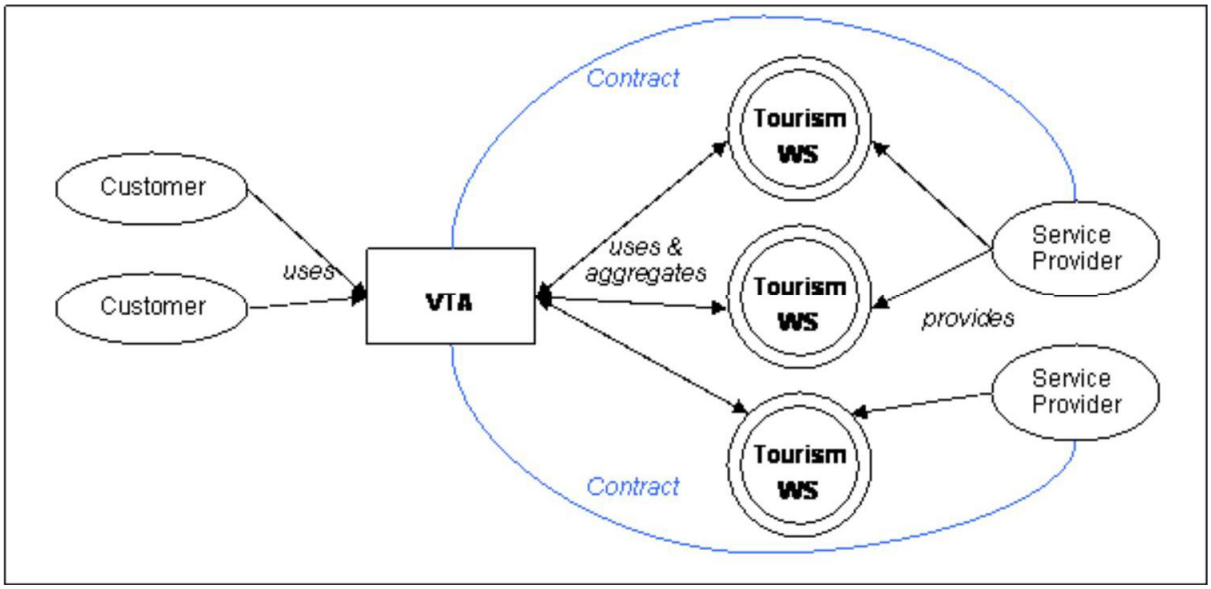


Fig # 14 Use case diagram

Actors, Roles and Goals
In the general use case there are 3 actors. The following defines why they participate in this use case (goal) and the particular interactions they are involved in (roles).

Customer:
The end-user that requests a service provided by the VTA
- Goal: automated resolution of the request by a user-friendly tourism service
- Role: end-user, interacts with VTA for service usage, payment, and non-computational assets (e.g. receiving the actual ticket when booking a trip)

Tourism Service Providers:
Commercial companies that provides specific tourism services
- Goal: sell service to end customers, maximize profit as a commercial company
- Role: provides tourism service as a Web Service (also provides the necessary semantic descriptions of the Web Services), may have a usage and allowance contract with the VTA

VTA:
The intermediate between the Customer and the Tourism Service Providers. It provides tourism services to customers by aggregating the separate services provided by the single Service Providers.
- Goal: provide high-quality end-user tourism services, uses existing tourism services and aggregates these into new services, maximize profit as a commercial company / represent union of service providers (depending on the owners of the VTA).
- Role: interacting with customer via user interface (can be web-based for direct human customers interaction or via Web Services for machine-users), usage and allowance contract

for Web Services offered by Service Providers, centrally holding all functionalities for handling Semantic Web Services (mechanisms for discovery,composition, execution, etc.)

8.2.Class Diagram

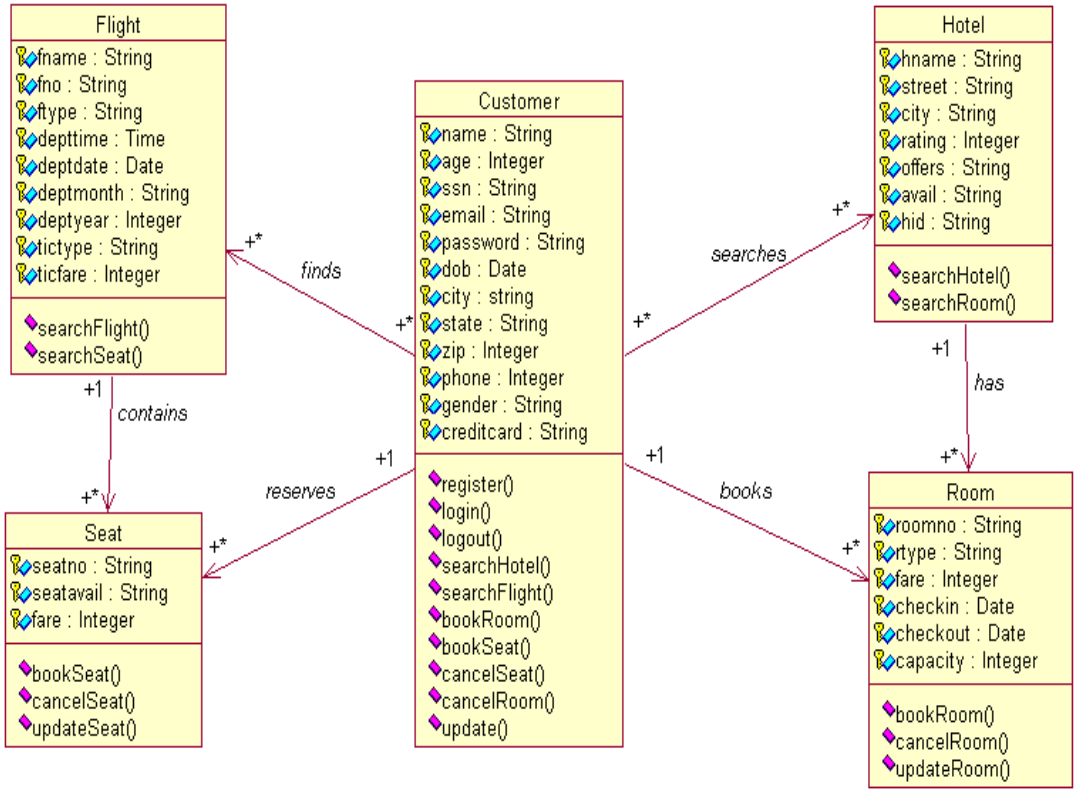


Fig # 15 Class Diagram

8.3. ER Diagram

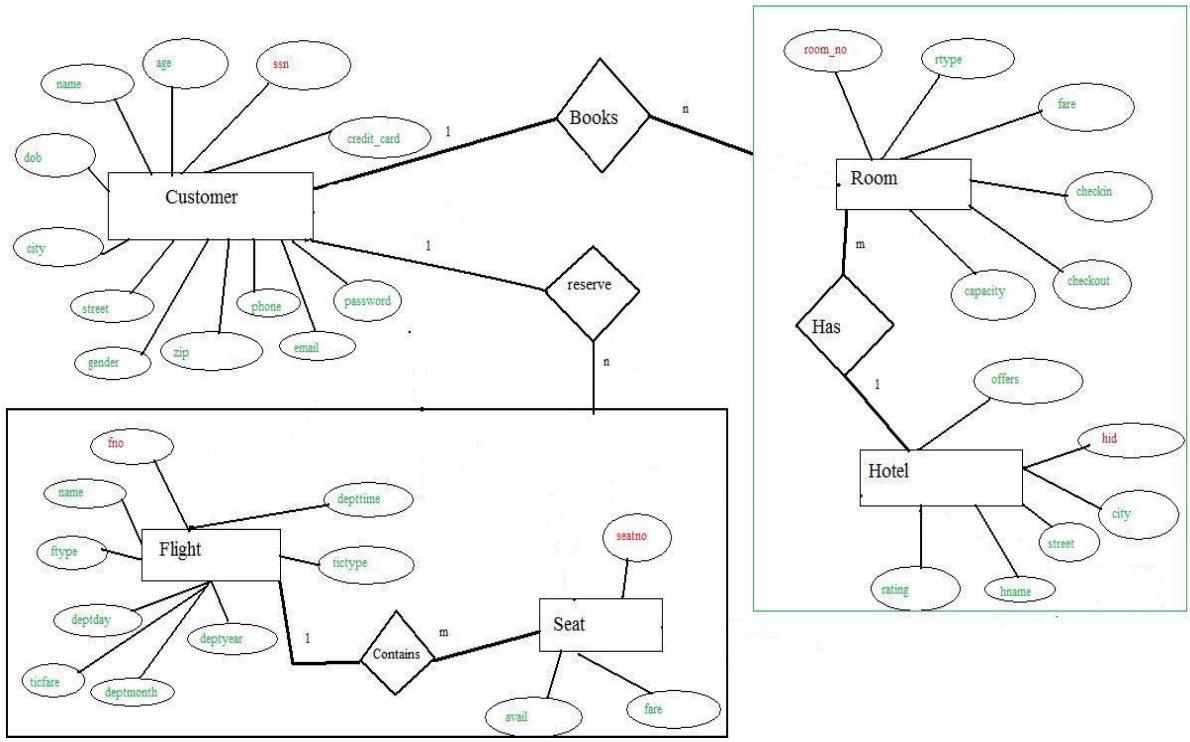


Fig # 16 ER Diagram

8.4.Sequence Diagram

8.4.1 General Flow

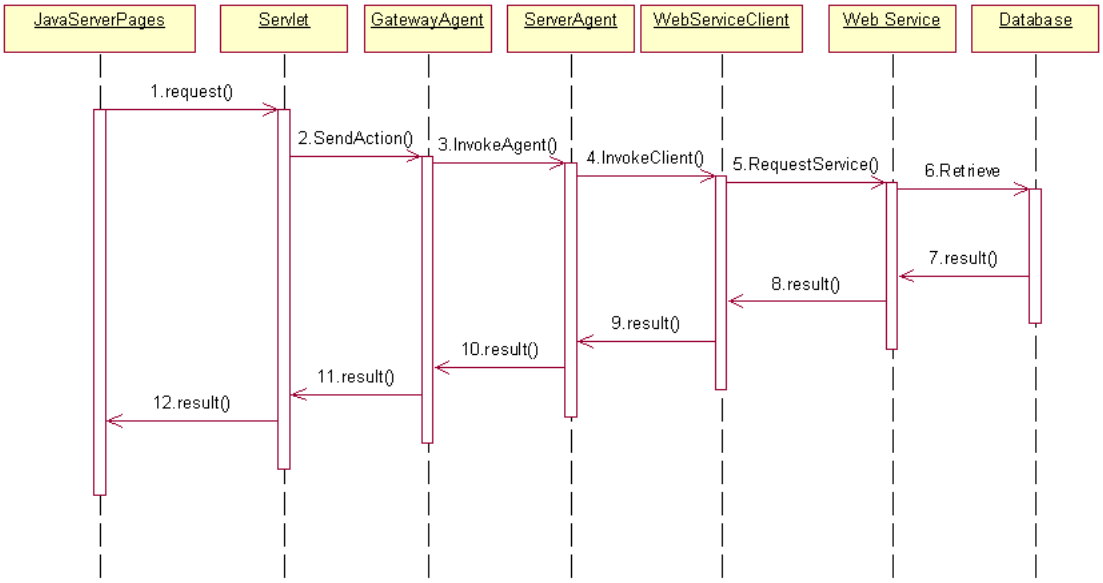


Fig # 17 General Flow

8.4.2 Registration

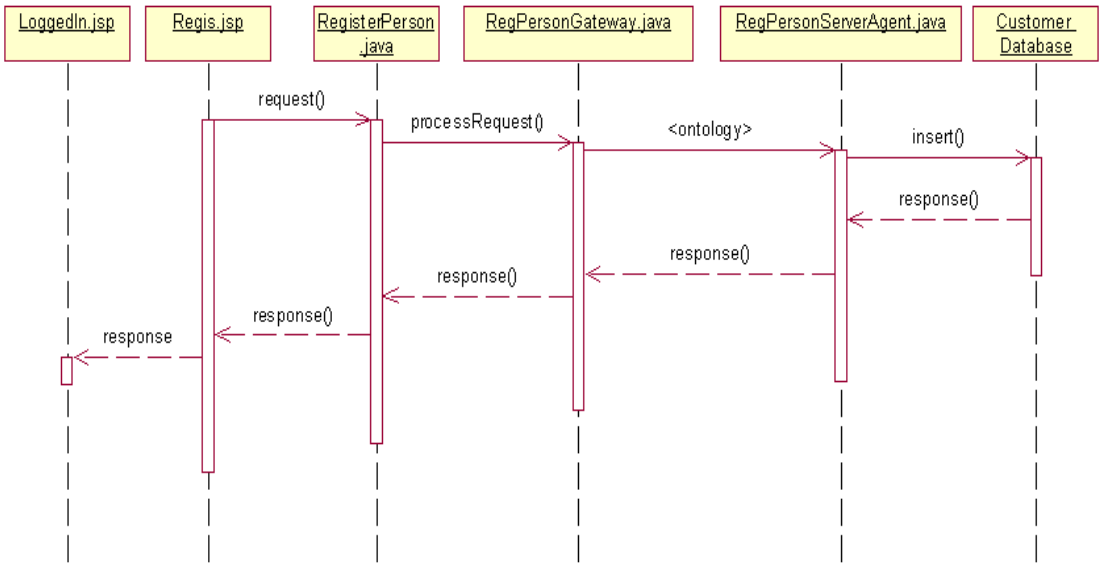


Fig # 18 Registration

8.4.3 Hotel Search

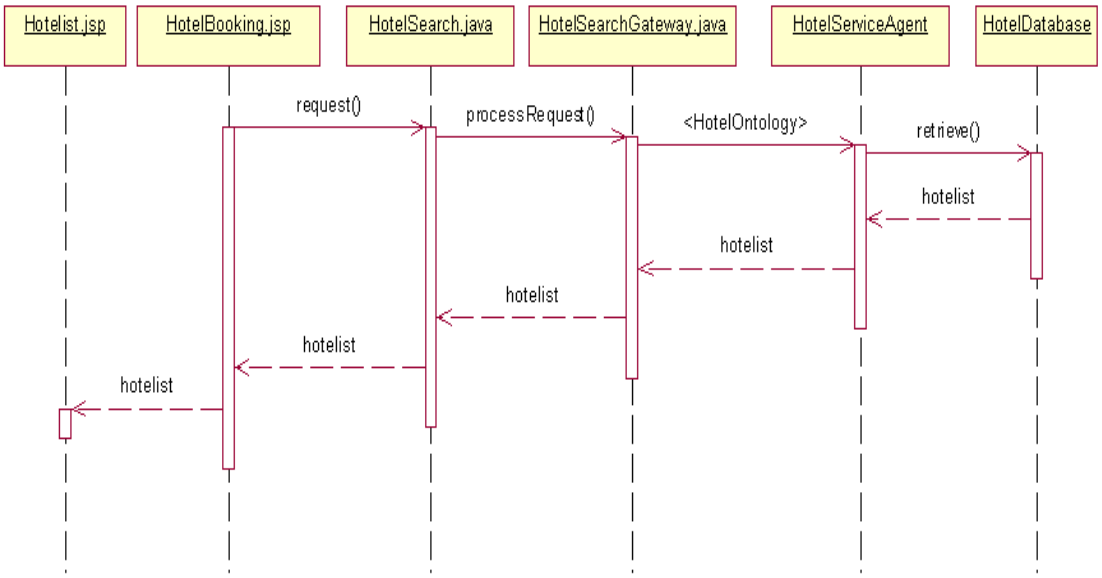


Fig # 19 Hotel Search

8.4.4. Hotel Booking

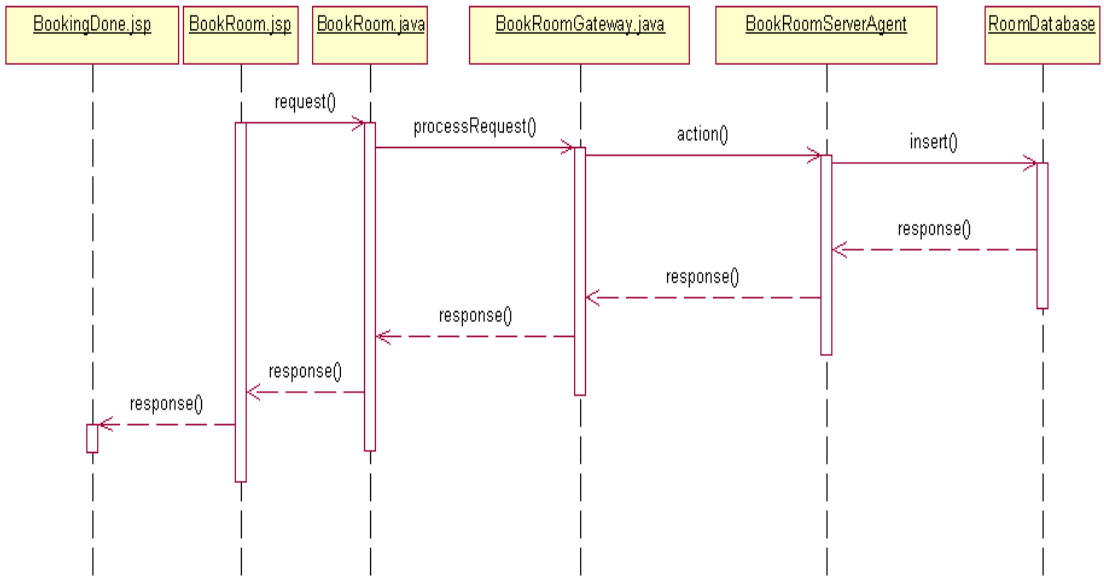


Fig # 20 Hotel Booking

8.4.5. Login

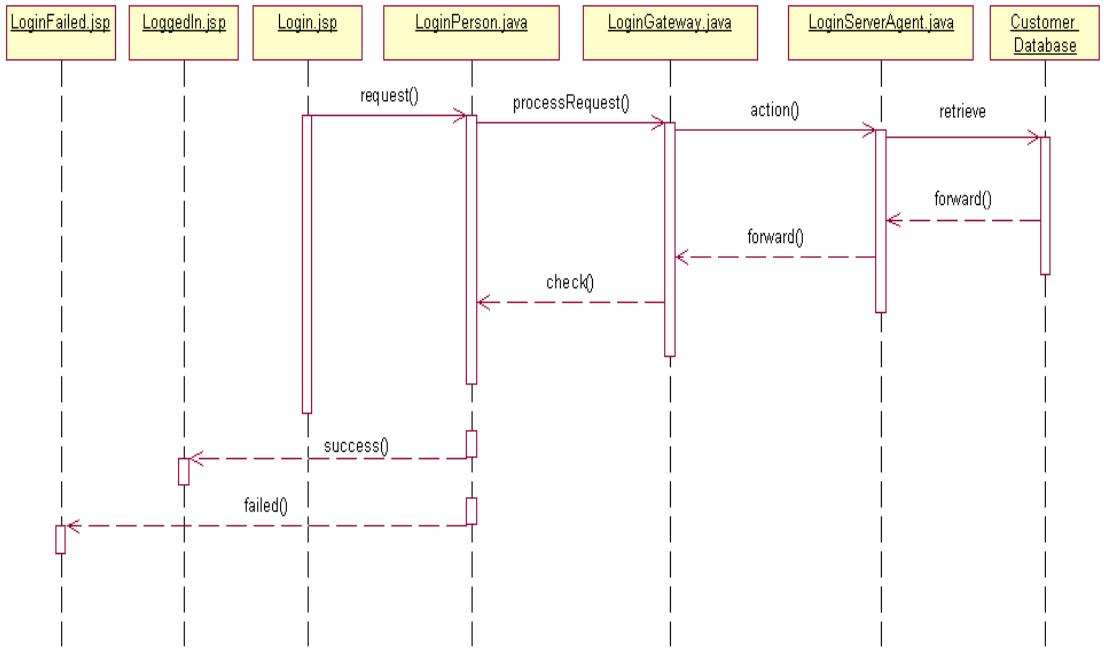


Fig # 21 Login

9.PROJECT ARCHITECTURE

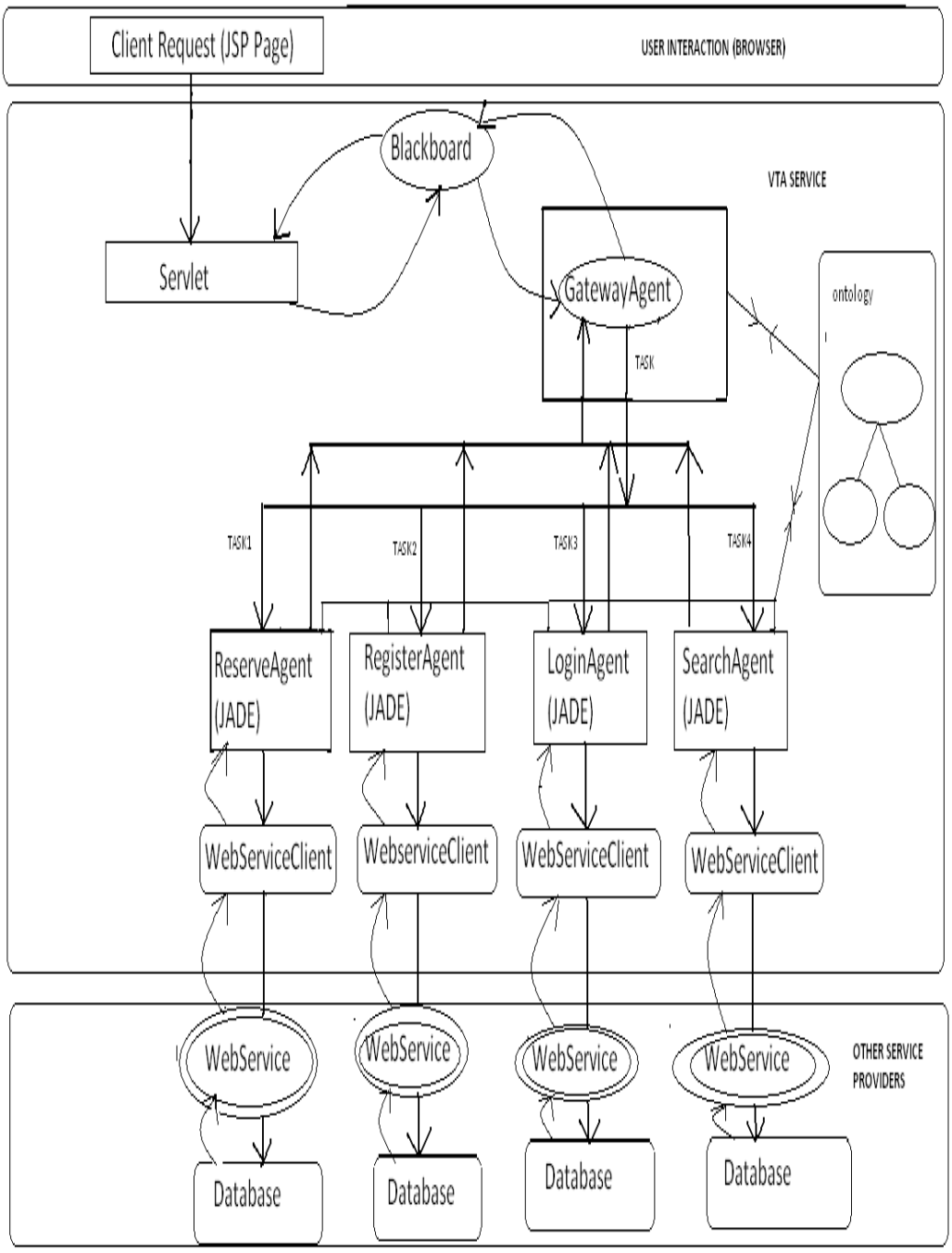


Fig # 22 Architecture

FLOW

Step 1.
In the browser the user causes an event and it generates a POST message.

Step 2.
The servlet handles it and the sendmessage() action is invoked.
The action creates a new BlackBoard object which will be the message channel between the GatewayAgent and the servlet.

Step 3
The GatewayAgent gets the dashboard object created previously and extracts who is the recipient and what's the message.After that it sends the message.

Step 4
ServerAgent who is now the recipient and responses to the GatewayAgent.

Step 5
ServerAgents referring to the ontological description invoke appropriate client web service.

Step 6
That in turn invokes the web service on the service providers domain that retrieves result from its database and passes it on.

Step 7
When it reaches the GatewayAgent,the GatewayAgent packes the reply and sends it via BlackBoard to the servlet. The servlet forwards it to the browser.

10.HARDWARE AND SOFTWARE REQUIREMENTS

SOFTWARE REQUIREMENTS:

1. Netbeans IDE 7.0 or higher
2. Java Agents Development Environment(JADE)
3. JDK 1.6 or higher
4. Derby Database

MINIMUM HARDWARE REQUIREMENTS

Processor	RAM	Disk Space
Intel Pentium III or AMD - 800 MHz	128MB	100MB

RECOMMENDED HARDWARE REQUIREMENTS

Processor	RAM	Disk Space
Intel Pentium Dual Core or AMD - 1.2 GHz	512MB	512MB

11.SCREEN SHOTS

11.1.Home page

Fig # 23 Home Page

11.2.Register

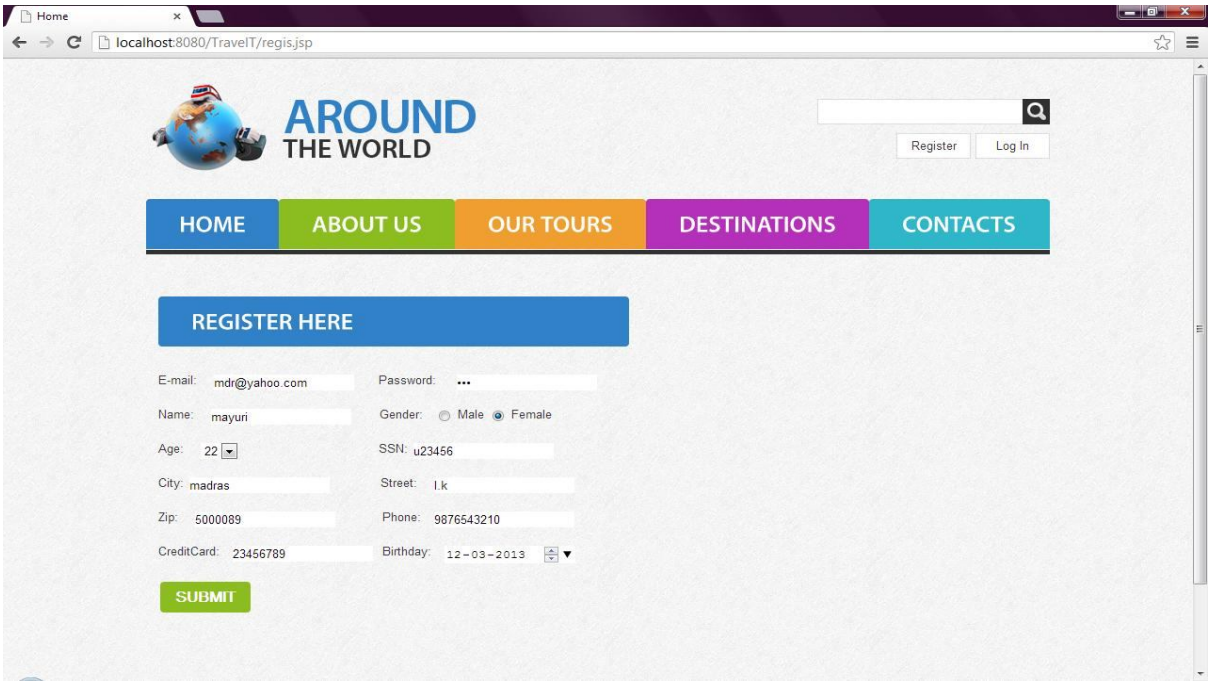


Fig # 24 Register

11.3.Hotel Booking page

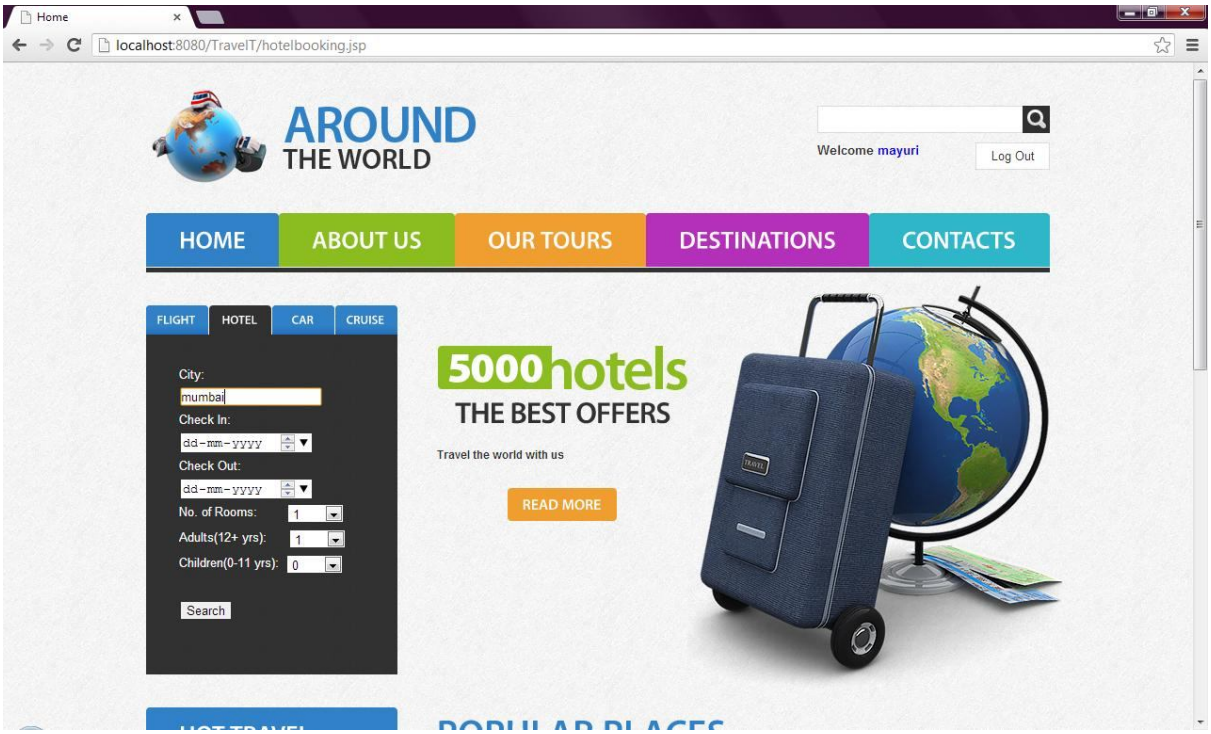


Fig # 25 Hotel Booking

11.4.HotelList page

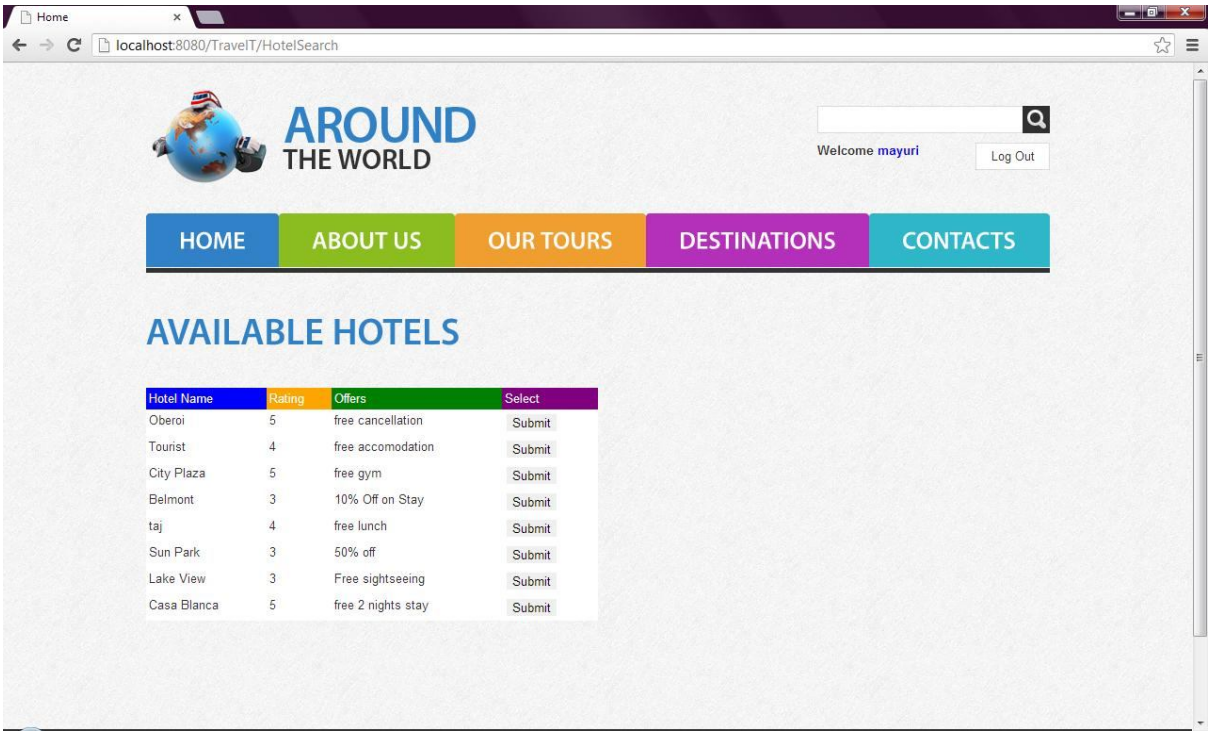


Fig # 26 Hotel List

12.FUTURE SCOPE

Apart from Hotel and Flight services, this project can be extended to provide other services like **Car, Cab, Cruise, Train** etc.

This approach may be applied in numerous domain search requirements,
such as in **e-learning,** the system can be used for searching, selecting and registering courses/modules. By finding, selecting and combining these services, the system can decrease the user interaction in these services for e-learning management.

It can be applied to all e-commerce applications.

13.CONCLUSION

To conclude, current Web Service discovery mechanisms do not support our requirements well enough.

So this project introduces an architecture for e-commerce applications with Semantic Web technologies, which is very suitable for automated service. The use of Semantic Web Service will enable automatically service finding and composition.

The Agents are assigned the task of finding an appropriate Server Agent to invoke the service depending upon the parameters that has been provided by the Client-end, thus avoiding the static binding and invocation problem found in web-services.

Agents being independent entities make it an easier task to impart the system scalability and to some extent proving it to be a bit more reliable as well. Thus providing ,partially, a solution to the discovery problem.

Thus one of the many solution that has been proposed to implement a Semantic Web Service has been implemented in our project.

14.REFERENCES

[1] JADE TUTORIAL-
JADE PROGRAMMING FOR BEGINNERS.
http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial- for-beginners.pdf

[2] JADE TUTORIALS- simple example for using the jadegateway class.
http://jade.tilab.com/doc/tutorials/JadeGateway.pdf

[3] JADE TUTORIALS- application defined content languages and ontologies.
http://www.cs.uta.fi/kurssit/AgO/harj/jade_harkat/doc/CLOntoSupport.pdf.

[4] Sandeep Kumar, R.B.Mishra. Multi-Agent Based Semantic Web Service Composition Models

[5] Srividya Kona,Ajay Bansal,Gopal Gupta . Automatic Composition of SemanticWeb Services .In ICWS, '07

[6] Software Agents. http://en.wikipedia.org/wiki/Software_agent.

[7] SWS Tutorials,STI,Innsbruck

[8] Introduction to Ontologies-Techwiki.
http://techwiki.openstructs.org/index.php/Intro_to_Ontologies

[9] D3.3 v0.1 WSMO Use Case "Virtual Travel Agency"
(http://www.wsmo.org/2004/d3/d3.3/v0.1/)

[10] Semantic Web Service Oriented Model for E-Commerce
(Institute of Scientific and Technical Information,China)

[11] www.wikipedia.com

15.APPENDIX

List Of Figures

List Of Codes