Distributed Service Provider Simulation using CORBA

Visvasuresh Govindaswamy, Saurabh Maitra, Mitvin Shah, Viswanath Veerappan
University of Texas at Arlington
416 Yates Street
Box 19015
Arlington, TX
76019-0015

victor@uta.edu, maitra@cse.uta.edu, mshah@cse.uta.edu, vxv7378@omega.uta.edu,

Abstract

This paper presents the current and future work on a Distributed Service Provider Simulation (DSPS) using CORBA that keeps track of available parking slots and tables in restaurants in a dynamic fashion. The communication architecture of the DSPS simulator is based on the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standard. Starting with an overview of the project requirements, the CORBA middleware, a comparison study between CORBA and the other alternative, Distributed Component Object Model (DCOM), and the software modules of the DSPS environment are described with respect to project requirements. With the DSPS simulating restaurant reservation system, performance measurements evaluating critical design and implementation decisions are described. The main aspects of the performance analysis are the attained application performance using CORBA as communication middleware, and the scalability of the overall approach. The evaluation shows the appropriateness of the design of the DSPS environment and the derived software architecture, which is flexible and open to further extensions. Moreover, CORBA provides a suited platform for distributed interactive simulation purposes because of the adequate performance, high scalability, and the high-level programming model, which allows rapidly developing and maintaining complex distributed applications with high-performance requirements.

Keywords: *CORBA, Distributed Service Provider Simulation (DSPS), Distributed System, Middleware.*

1. Introduction

The ever-changing dynamic and sudden growth of today's networks has increased the need of accessing network services that are provided by servers. This has led to the development of two important requirements [1, 2]. It has demanded the growth of service providers to provide services for the clients who request these services. As a result, it has become necessary for a client to request for certain services with some kind of modality such as payment and/or authentication in order to possess such services from these service providers. The reliance of clients have increased dramatically on these service providers that it has become necessary for the service provider to be distributed, reliable, scalable and powerful enough to service incoming requests at an acceptable performance level. As for the client, it is ideally for it to be general-purpose in nature. All it has to possess is an engine to allow it to access the service provider. Hence, a service provider is a host that provides a set of clients with a certain number of remote services.

This paper presents a Distributed Service Provider Simulation (DSPS) that keeps track of available parking slots and tables in restaurants in a dynamic fashion. The following scenario has been considered:

The central business area of a city has several restaurants and parking places distributed geographically. It's very hard to get a parking slot and a table during lunchtime. It's even harder to get a parking slot close to a restaurant with available tables. Imagine a 'service' on the distributed computing environment that keeps track of available parking slots and tables in restaurants in a dynamic fashion. A user can contact this 'service' through his palm device (or cell phone). Basically the user tells the service where she/he is, what kind of cuisine she/he likes and a time limit. The 'service' processes the request and finds a parking slot – restaurant match so that the distances (driving and walking) and waiting time are minimized. The scenario assumes availability of GPS enabled devices, PCs, cameras, sensors etc.

The paper is organized as follows: in Section 2, the past literature on CORBA and service providers is reviewed. In Section 3, the CORBA middleware is presented, which is followed by a comparison study between CORBA and the other alternative, Distributed Component Object Model (DCOM) in Section 4. In section 5, the software modules of the DSPS environment are described with respect to project requirements. Section 6 discusses the distributed aspects of the design. Section 7 focuses on performance measurements evaluating critical design and implementation decisions, and analysis. The conclusion is presented in Section 8 with current work, along with directions for future work.

In this paper, the reader is assumed to have a basic knowledge of the fundamentals of the object-oriented paradigm, the Java programming language [11, 13] and CORBA [10, 12].

2. Literature review

This section looks into past research into service providers and/or research that are done at the crossing of the two main technology streams of the Java and object-oriented technology, especially Java and CORBA or CORBA-like systems.

2.1. Service Providers

Carchiolo et.al [1] present an approach based on mobile agents for the implementation of a platform for an Internet service provider. The aim of the approach is to implement a service provider that can provide services that are reliable, low-cost and flexible. The main emphasis here is to provide clients who do not have particular hardware characteristics and do not need special software to access the service they require but are only equipped with an engine for network access.

2.2. CORBA vs. DCOM

Janson [3] in his thesis, studies in detail the differences between Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) and Microsoft's Distributed Component Object Model (DCOM). In order to make the right choice between these technologies, the authors describe both technologies thoroughly and compare them along with practical performance tests. The ease of deployment was also considered.

3. ORB Architecture

The conceptual architecture [5, 6, 7, 8, 9] of an ORB is shown in figure 1. The CORBA communication middleware provides the transparency and the information passing technology. The object implementation is made available through the server skeleton, which is used by the object adapter to route incoming requests to their implementations. The object adapter (OA) is responsible for providing basic functionality for objects and servers, such as processing object references, activation and deactivation of objects and method invocations. An implementation can use the OMG-defined Basic Object Adapter (BOA), or use a custom OA for specialized purposes such as accessing database objects. The ORB Interface provides access to the interface and implementation repositories as well as general functionality such as the conversion of objects to strings and vice versa. This interface is identical for all implementations and can be accessed by clients as well as servers. Clients issue requests either through client stubs, the Interface Definition Language (IDL) or through the Dynamic Invocation Interface.

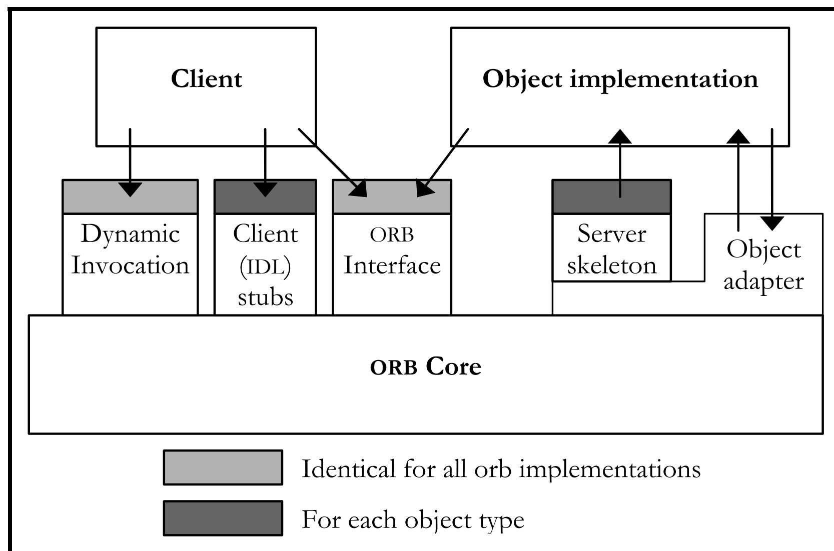

Fig. 1. The Structure of ORB Interfaces

4. Comparison between CORBA and DCOM

This subsection lists the main differences [3, 14, 15] between CORBA and DCOM.

Table. 1. CORBA vs DCOM

	CORBA	DCOM
Architecture	Dominant remoting architecture	Dominant component architecture
Strategy of Implementation	Horizontal, since OMG aim to create portable distributed applications for many different vendors' platforms.	Vertical, since Microsoft wants to control the technology from all the way from the operating system up to the end-user applications
Performance	Similar to each other.	
Programming	CORBA products for several	Applications are mostly developed in

Languages	languages, for instance Java, C/C++, Smalltalk, COBOL and Ada.	C++ and Visual Basic, and some developers use J++.
Learning Curve	If a programmer knows Java, it is easier to learn to use COM+ than CORBA, since the COM+ code is almost like ordinary Java while CORBA adds special code. Building a large distributed application with CORBA requires more effort and knowledge from the programmer.	
Development Tools	Tools are less sophisticated	Tools are more sophisticated
Platforms	Any platform	Run on Windows and only Windows

5. Project Implementation Details

The hardware and software details are given in sections 5.1 and 5.2 respectively and section 5.3 covers the architecture used in the project.

5.1. Hardware Details

Machine: Gamma - SUN Ultra Enterprise 3000 system
Processor: UltraSPARC at 248 MHz
Ram: 4 GB
Connection: 100 Mbps Ethernet

5.2. Software Configuration Details

Programming language: Java
Operating system: Sun Solaris
CORBA ORB: Orbix 2.3c

5.3 Architecture of project

Distributed Service Provider Simulation (DSPS) using CORBA represents a distributed system that keeps track of available parking slots and tables in restaurants in a dynamic fashion. Figure 2 shows an overview of the DSPS environment.

5.3.1 Software Modules

The architecture identifies the following modules to represent each functional unit.

- Domain Server – The server is a one-point contact to the client and coordinates the overall process. It communicates with the other components like parking lots and restaurants registered with it and other domain servers to respond to the requests from a client
- Parking Lot and Restaurant – These objects simulate the functioning of restaurant and parking lot. Each such object is registered with one domain server.
- GPS and Sensor – The GPS object tracks the location of the client. The sensor objects continuously monitor the availability of slots/tables in the parking lots/restaurants and duly inform the respective objects using callback functions, which in turn, keep the server updated.

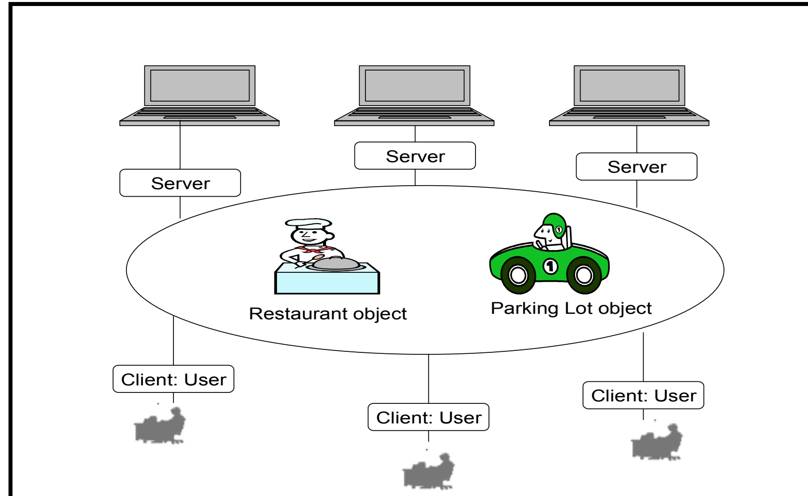

Fig. 2. Overview of the DSPS environment

5.3.2 General Working

The client, that is, the user informs the service where she/he is, what kind of cuisine she/he likes and a time limit. The server processes the request and finds a parking slot – restaurant match so that the distances (driving and walking) and waiting time are minimized.

The entire business domain is divided into several domains. A server represents each of the domains. When the request arrives on the server side, each server checks whether the coordinates and preferences fall within its domain. The server considers the objects in its own domain and tries to find the nearest restaurant and parking lot as per the requirements. In the event that the server fails to find any restaurant, the computation is migrated to other domain servers by forwarding the client requirements. Upon finding a restaurant/parking lot as per the request, the domain server makes a reservation at the respective functional unit by invoking their update methods. Multiple clients are serviced by making the domain severs multi-threaded.

The working of the system can be understood better with the sequence diagram shown in Figure 3.

5.3.3. ORB Interface Definition

The data structure and the interface definitions in the IDL file used in the implementation are given in Figure 4.

6 Distributed system features in the DSPS architecture

This section discusses aspects of the DSPS architecture vis-à-vis the features required of a distributed system.

6.1 Information sharing

Information sharing is required in the system to overcome the unavailability of restaurant or parking lot in the same domain as the client, as per the client's needs. In the design, it is realized between domain servers whenever a client request is forwarded from one domain server to others in order to find the nearest restaurant and parking lot. It is also achieved when one server becomes overloaded and it has to transfer the request to a nearby server. The information shared in these cases is the client's location and requirements and necessary domain data.

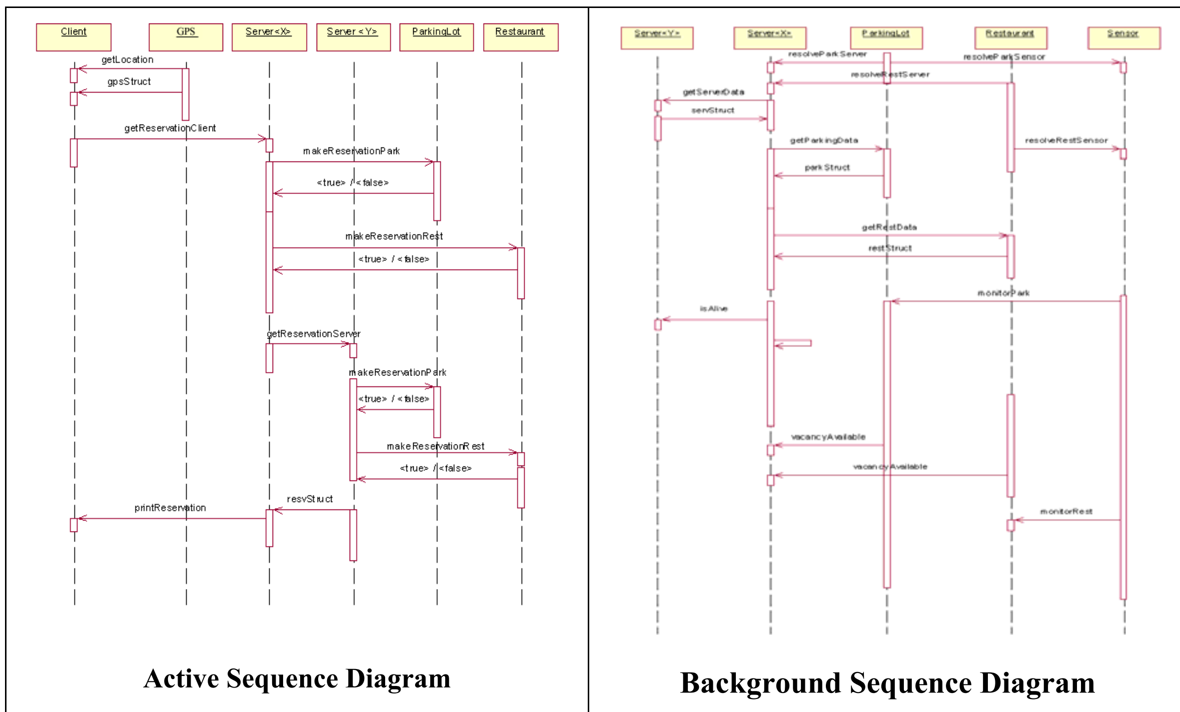

Fig. 3. Sequence diagrams for the design

6.2 Transparency

The architecture provides location and performance transparency by ensuring that client, at all times, is unaware of the identity or location of the domain server that services its request.

6.3 Scalability and Flexibility

Scalability [2] of a system is the ability of the system to be able to find a solution that works when the size of the problem grows. Flexibility can be interpreted to be the ability of a system to adapt to dynamically changing situations. In this project, a domain server can register any number of parking lots and restaurants. Multiple client requests can be handled due to the multithreaded nature of the domain servers. The flexibility in the system is limited due to the fact that the server does not keep track of the client after the request is serviced.

6.4 Fault tolerance

The current design does not provide fault tolerance in the event of server/functional unit crashes. However, it is proposed that it can be introduced in the system by checking the alive status and the subsequent distribution of domain data and computation to other servers. It is also proposed to include active exception handling and recovery mechanisms by providing call back methods on the other domain servers to regain the lost information and continue servicing client requests at the just failed domain server. The client, which had been in contact with a failed server, will catch an exception and get the reference to another domain server for further communication.

```
module Business
{
  exception BusinessException
  {
    string reason;
  };

  struct gpsStruct
  {
    short x;
    short y;
  };

  struct resvStruct
  {
    string restName;
    string parkName;
    short rA;
    short rT;
    short pA;
    short pT;
    boolean resvAvail;
  };

  struct restStruct
  {
    string name;
    short x;
    short y;
    string menuItem;
    short totalVac;
    short availableVac;
  };

  struct parkStruct
  {
    string name;
    short x;
    short y;
    short totalVac;
    short availableVac;
  };

  struct clientStruct
  {
    string name;
    short x;
    short y;
    string menuItem;
    short time;
  };

  struct servStruct
  {
      string name;
      short numServersUp;
      short x;
      short y;
  };

  interface IntServerClient
  {
     void printReservation(in resvStruct reservationData);
  };

  // interface to be used by client to get its location
  interface IntClientGPS
   {
        // returns location data polled by client
        gpsStruct getLocation();
   };

  // base interfaces to be used by restaurant
  // and parking lot interfaces
  interface IntServerRest
  {
     // returns data polled by server
     restStruct getRestData();

     // returns bool from rest, parking lots confirming reservation
     boolean makeReservationRest();
  };

  interface IntServerPark
  {
    // returns data polled by server
    parkStruct getParkingData();

    // returns bool from parking lots confirming reservation
    boolean makeReservationPark();
  };

  interface IntSensorPark
  {
      void monitorPark();
  };

  interface IntSensorRest
  {
    void monitorRest();
  };

 // Parking Lot and Restaurant interfaces
 interface IntPark:IntServerPark,IntSensorPark
 {

 };

 interface IntRest:IntServerRest,IntSensorRest
 {

 };

  // base interfaces to be used for sensor interface
  interface IntParkSensor
  {
     void resolveParkSensor(in IntSensorPark lotRef);
  };

  interface IntRestSensor
  {
      void resolveRestSensor(in IntSensorRest restRef);
  };

  // Sensor interface
  interface IntSensor:IntParkSensor,IntRestSensor
  {

  };

 // methods available at server, can be called by Restaurant
 // ParkingLot or other server objects
 interface IntServerServer
 {
     // function used by domain server to poll data from
     // other domain servers
 resvStruct getReservationServer(in clientStruct cData,in resvStruct rSt);
 servStruct getServerData(in IntServerServer sRef,in servStruct sSt);
 };

 interface IntClientServer
 {
     // sends request for reservation to domain server from client
 void getReservationClient(in clientStruct cData, in IntServerClient cObj);

 };

 interface IntVacancy
 {
     // Parking Lot and restaurant objects to inform status
   void vacancyAvailable(in boolean vacAvailable, in string id);
 };

 interface IntRestServer:IntVacancy
 {
   // callback to resolve restname
   short resolveRestServer(in IntServerRest restRef);
 };

 interface IntParkServer:IntVacancy
 {
   // callback to resolve parkname
   short resolveParkServer(in IntServerPark lotRef);
 };

 interface IntServer:IntParkServer,IntRestServer,
                     IntClientServer,IntServerServer
 {
   // returns live status of this server to other servers
   boolean isAlive();
 };
};
```

Fig. 4. IDL file for the implementation

7 Introduction of the Tests

This section discusses the metrics and tests that are being applied to the simulation.

7.1 Calculation of Response Time

The response time is calculated as follows: Time t_1 is the time just before the client makes a request while t_2 is the time for the client to receive an answer from the server for that particular request. The difference between t_1 and t_2 is the response time for that particular request. Hence,

$$t_2 - t_1 = t_{OverheadSend} + t_{Communication} + t_{OverheadReceiveRequest} + t_{Execution} + t_{OverheadReturn} + t_{Communication} + t_{OverheadReceiveResult}$$

$t_{OverheadSend}$: Time required for marshalling the input parameters and putting the request on the “wire”,

$t_{Communication}$: Time required on the wire for the client to send the request to the server and vice versa,
$t_{OverheadReceiveRequest}$: Time required for receiving the request and unmarshalling it.
$t_{Execution}$: Time required for executing on the server
$t_{OverheadReturn}$: Time required for marshal the result and putting the result on the "wire",
$t_{OverheadReceiveResult}$: Time required for receiving the result and unmarshalling it.

7.1.1. Assumptions for the tests

Since the system is implemented using the Java language, the resolution of the clock is limited to milliseconds. This implies that the difference between times t_2 and $t_1$is too small to be measured over one invocation. Thus it is necessary to take an average over multiple invocations. The clock is started when the Java method System.currentTimeMillis( ) is called before the first invocation and is stopped after the nth invocation. The collected time is then divided by n to get the average roundtrip time for one invocation. The following techniques [13] have been used and will be considered when calculating the roundtrip time for these tests:

- The Java method System.currentTimeMillis( ) takes some time when it is called by another function. This will cause an increment in the response time when measuring over a period of time. This increment needs to be subtracted from the response time.
- The time it takes for a loop containing multiple requests need to be considered and measured. It should also be subtracted from the response time.
- When a method is invoked on a servant for the first time; it will take a longer time since there is an overhead. Measurements should be taken after the first invocation, thereby eliminating this overhead.

7.1.2. Multi Clients

The objective here is to detect contention by varying the number of clients sending their requests to the same server. Their response is collected and compared. All the clients call the same servant since the domain servers; spawn a new thread for servicing each request by the client, Figure 5 shows a schematic model of this scenario.

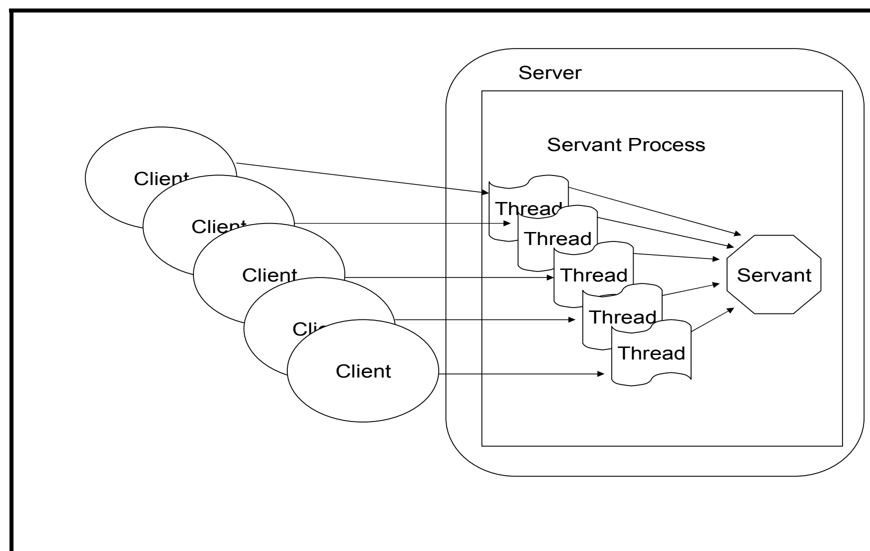

Fig. 5. Schematic model of the Multi Client Test

7.2. Results

This section shows the preliminary results of two scenarios obtained during scalability tests.

7.2.1. First Scenario

Here, the response time is calculated by scaling the number of Parking Lot objects while keeping the number of Servers to 4 and the Restaurant objects to 1. The results are shown in figure 6. It shows that the rate of increase decreases as the number of Parking Lots is scaled up.

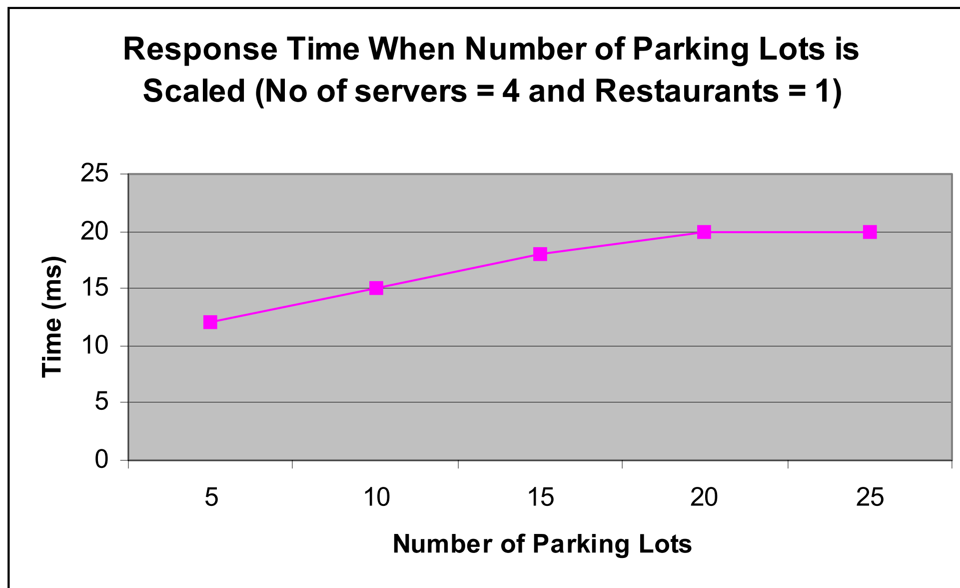

Fig. 6. Response Time when number of Parking Lots is scaled up

7.2.2. Second Scenario

Here, the response time is calculated by scaling the number of Restaurant objects while keeping the number of Servers to 4 and the Parking Lot objects to 1. The results are shown in figure 7. It shows that the rate of increase decreases as the number of Restaurants is scaled up.

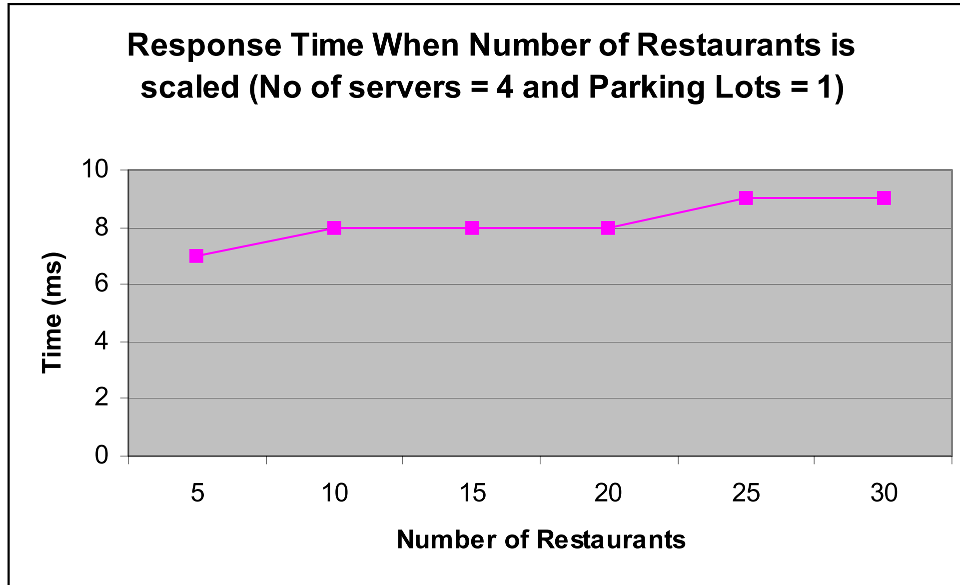

Fig. 7. Response Time when number of Restaurants is scaled up

Description of procedure

1. Each client calculates the start time.
2. Each client calculates the elapsed time.
3. Each client displays the response time.

8. Conclusion, Present and Future Work

The need for simulation frameworks to be transparent, portable and extensible makes the choice of distributed object technology and a good design to be crucial. Location transparency in the architecture is provided by the fact that the client is unaware of the server from which the response is obtained. Performance transparency is achieved by forwarding client requests from an overloaded server to the next nearest server that can continue with the computation. The distributed service provider implemented in this project shows that the middleware used for the system has a major role in deciding how well the system performs in terms of transparency, scalability, response time and other factors. The current design is scalable as the parking lots and restaurants can be dynamically registered with the domain servers. Also, multiple clients can be serviced simultaneously as the domain servers are multithreaded. Thus, the service provider in this project has provisions for the parking lots and restaurants to register with their respective servers and the client can issue requests to the servers. Performing transactions, handling of errors and server activations are some of the other issues that can be handled using CORBA. Future work is to include improvement in fault tolerance of the system and dynamic load balancing between servers by introducing more functions in the server interface of the IDL file. Error handling procedures and security issues are also being investigated in order to improve the robustness of the implementation.

9. References

[1] V. Carchiolo, M. Malgeri, G. Mangioni, “An Agent Based Platform for to Service Provider”, in *Proceedings of 2nd IMACS International Conference on Circuits, Systems and Computers* (CSC' 98), Piraeus (Greece), October 26-27-28, 1998.

[2] S. Maffeis: “Adding Group Communication and Fault Tolerance to CORBA”, In: *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June 1995.

[3] F. Janson, and M. Zetterquist, “CORBA vs. DCOM”, *Master Thesis, Technical Report*, The Royal Institute of Technology, Kungliga Tekniska Högskolan.

[4] Object Management Group: *OMA: Object Management Architecture Guide, Revision 2.0.* http://www.omg.org

[5] Object Management Group and X/Open: *The Common Object Request Broker: Architecture and Specification, Revision 1.* http://www.omg.org

[6] Object Management Group: *Common Object Services Specification, Volume I.* http://www.omg.org

[7] Object Management Group: *Common Object Request Broker: Architecture and Specification, Revision 2.0.* http://www.omg.org

[8] Object Management Group: *ORB Portability Enhancement RFP*. OMG Document number 95-6-26. http://www.omg.org

[9] Object Management Group: *Common Facilities RFP3 -- Data Interchange Facility and Mobile Agent Facility*. OMG Document Number 95-11-3. http://www.omg.org

[10] Jon Siegel et al.: *CORBA Fundamentals and Programming.* John Wiley & Sons, 1996.

[11] Sun Corporation: *The Java Programming Language.* http://java.sun.com

[12] J. Buford: *CORBA and WWW: On a Collision Course*. (position paper) *Joint W3C/OMG Workshop on Distributed Objects and Mobile Code*. Boston, June 1996.

[13] A. Kari, L. Casper and S. Reijo. “Process Enactment Support in a Distributed Environment”, *WET ICE '95, IEEE Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Berkeley Springs, West Virginia, April 20-22, 1995.

[14] P. Chung, H. Yennun, S. Yajnik, D. Liang, J. Shih, C. Wang, Y. Wang. “DCOM and CORBA Side by Side, Step by Step, and Layer by Layer”, http://www.cs.wustl.edu/%7Eschmidt/submit/Paper.html [2000, August 30].

[15] J. Pritchard. 1999. *COM and CORBA Side by Side*. ISBN: 0-201-37945-7. United States of America. Addison Wesley Longman, Inc.