An Analysis on CORBA
CHAPTER 3

AN ANALYSIS ON CORBA

3.1 Introduction

Distributed object systems, like CORBA, can reduce the cost of software development due to reusability. These systems provide reuse of data and code of procedures at a higher level, using components. It thus aids in the evolutionary software development. Component based development and component reuse are established practices. Component based software engineering [131] [132] brings forth reusability at a higher level. CORBA provides reuse of data and code of procedure at a higher level. Reuse at binary level is efficient but complicated when compared to reuse using inheritance at a higher level, since it uses whole procedure reuse, procedure parameterization and reuse of statically allocated data [133].

A methodology referred to as built-in contract testing [134] checks the pair wise interactions of components in component-based software construction at integration and deployment time. This technology is integrated with and made to supplement the entire development cycle starting from requirements specification activities and modeling. A generic mechanism for component connection [135] uses connectors from producer component to consumer component. In these methods, the interaction functionality is not modeled as components and hence the connectors between the components cannot be reused. This chapter discusses about the assembly of components and links dynamically using application oriented approach, where the component functionality is separated from interactions among them using connector components.

Java and distributed object models are important for building modern, scalable and interoperable applications. Java RMI marshals data using Java object serialization and so is language dependent [136], while DCOM is platform dependent [137][138]. Although CORBA is language and platform independent, it suffers from communication efficiency. The performance
assessment framework for distributed object models, considers the performance criteria for all important aspects of distributed object computing, including single and multi-client scenarios. It presents the high and low level design of the framework and gives insights into implementation details for CORBA, RMI and RMI-IIOP. A performance assessment of RMI and CORBA has been carried out [139][140][141] and it is found that CORBA suffers from communication inefficiency. Separate benchmark suites [142] for CORBA brokers have been developed for broker vendor and user audiences. The vendor suite is a result of several benchmarking projects with industrial partners and covers the entire functionality of a CORBA broker. The user suite is simplified to give an overview of the basic factors influencing broker performance and is complemented with an approach for tailoring the results to a specific system and mode of operation without a prohibitive loss of precision. Using RTT, this chapter presents an analysis on the performance and overhead of four important distributed object models for Java, namely, RMI, RMI-IIOP, CORBA and DCOM.

3.2 Common Object Request Broker Architecture

The main components of CORBA [5] as shown in figure 3.1 are listed below:

1) Object Request Broker (ORB): ORB provides the communication and activation infrastructure for distributed object applications. The key feature of the ORB is the transparency of how it facilitates client/object communication. This transparency includes transparency to object location, object implementation, object execution state and communication mechanisms. To make a request, the client specifies the target object by using an object reference (OR). When a CORBA object is created, an OR is also created. It is unique, immutable and opaque. A client can obtain an OR on object creation, from directory service or on string-to-object conversion.

2) OMG Interface Definition Language (OMG IDL): An object’s interface specifies the operations and types that the object supports and thus
defines the requests that can be made on the object. Since OMG IDL is a declarative language, not a programming language, it forces interfaces to be defined separately from object implementations. This allows objects to be constructed using different programming languages and yet still communicate with one another. OMG IDL types are mapped onto programming languages like C, C++, COBOL, Java, SmallTalk and Perl.

3) **Interface Repository (IFR):** The CORBA Interface Repository (IFR) allows the OMG IDL type system to be accessed and written programmatically at runtime. So it is used in dynamic object invocation.

4) **Stubs and Skeletons:** OMG IDL language compilers generate programming language types, client side stubs and server-side skeletons. A stub is a mechanism that effectively creates and issues requests on behalf of a client, while a skeleton is a mechanism that delivers requests to the CORBA object implementation. They are interface-specific and dispatching through stubs and skeletons is called static invocation.

5) **IDL Compiler:** It is a tool which generates static stubs (Static Invocation Interface (SII)) and skeletons (Static Skeleton Interface (SSI)) from the interface definition.

6) **Dynamic Interface Invocation (DII):** The DII allows clients to generate requests at run-time, which is useful when an application has no compile-time knowledge of the interface it accesses. The DII also allows clients to make deferred synchronous calls, which decouple the request and response portions of two-way operations to avoid blocking the client until the servant responds. The DSI allows an ORB to deliver requests to servants that have no compile-time knowledge of the IDL interface they implement.

7) **Object Adapters (OA):** It serves as the glue between CORBA object implementations and the ORB. It is an object that adapts the interface of an object to the interface expected by a caller.

8) **Inter-ORB Protocols:** To support interoperability, CORBA proposes General Interoperable Protocol (GIOP) and Interoperable OR (IOR). GIOP defines the communication between ORBs in general terms. It describes a Common / Character Data Representation (CDR) format and message formats for sending requests and responses between ORBs. GIOP is
defined independently of any particular transport protocol in order to accommodate a wide range of networking infrastructures. Internet Interoperable protocol (IIOP) provides full-duplex, connection oriented communication channel through the TCP/IP protocol. IOR is a representation of a CORBA object that gives the client an ability to access that object, while hiding the details of that object's implementation and status.

9) **ORB Interface:** To decouple applications from implementation details, the CORBA specification defines an interface to the ORB. This ORB interface provides standard operations to initialize and shutdown the ORB, convert object references to strings and back, and create argument lists for requests made through the DII.

![Figure 3.1: The Common Request Broker Architecture](image)

10) **Implementation Repository (IMR):** IMR contains information that allows an ORB to activate servers to process servants. ORBs use this facility to activate servers on demand. The Implementation Repository maintains a table that associates “Server Name” with “Start Command” and “Activation Mode”. The “Start Command” is executed in order to start a server when needed. The “Activation Mode” is a qualifier that specifies whether a new
server instance should be started for every client or the clients share the services of a single server instance.

3.3 Component Reuse and Assembly in Distributed Object Systems

Component based systems deal with the development of applications using components. Components are language independent software modules which provide services. Each component has a specification, implementation and executable. Component based system design can control the costs of developing software products and boost their quality. This is primarily due to the reusability of components. Different component technologies like COM, CORBA, RMI and RMI-IIOP support software reuse. Component infrastructure also has versioning support for managing multiple views of code.

Basically there are two approaches for the reusability of components in component based systems. They are object oriented approach and application oriented approach. These approaches are described below:

a) **Object oriented approach**: Here the systems are developed as Component libraries which can be reused. But an object communicates with other objects by invoking their methods. Reassembling and reusing components is difficult in this approach because the objects have implicit dependencies on each other and on the outside services like the middleware which are used for communication. COM and CORBA bring about interoperability and language independency. COM and CORBA separate component implementation from specification but do not solve the problems associated with implicit dependencies. In this method, the components are implemented as objects and interactions between them are through message passing mechanisms. For distributed operations middleware is used.

b) **Application oriented approach**: This method uses assembly approach on composition of components using frameworks. It separates concepts into:
i) **Architectural view:** It defines the components in the system, their functionality and their interaction. Services and events can be used to model components and their interactions.

ii) **Component view:** It is concerned with component composition and their implementation.

iii) **Distributed object infrastructure view:** It deals with achieving communication among components using Quality of Service (QoS) parameters like performance, reliability and security.

To encapsulate the component implementation or functionality and to separate it from interconnection mechanisms, ports and links are used. Ports are component functions and links are used for communication. A component with the port is fully encapsulated and is ready to be assembled. It is not concerned with dependencies outside the boundary namely, the component communication and middleware. A port instance is a property of the component. So it is easy to replace/customize. Links are reusable, replaceable and customizable as they are designed as components. This method thus provides a plug and play capability and a separation of the interaction between components and its core functionality.

### 3.3.1 Interaction of Components

A component interacts with other components in four basic ways. A component can provide services to other components, get services from other components, generate events and observe events. So the component has four boundary elements as shown in figure 3.2. They give rise to three types of component interaction styles as listed below:

- **Service-Service:** Here a component uses the services provided by another component;
- **Event-Event:** Here a component observes an event generated by another;
- **Event-Service:** Here an event generated by one component triggers a service provided by another component.
3.3.2 Port and Links

The port and link abstractions are used to bridge the gap between components and objects and between component interaction styles and object method invocations. Ports are communication agents and connection points at the component's boundary. They support all four component boundary elements and thus support the component modeling concepts in software architecture. Links separate component core functions from inter-component communication mechanisms and strategies. Links directly support all three component interaction styles. So component developers can delay making decisions about the link services until assembly or runtime.

Each component boundary element is associated with a port type. A port instance belongs to only one component, and each port has an attribute that holds the links connecting it. Ports can be service port or event port as shown in figure 3.3. Service ports have properties that describe the services they provide or require. Event ports have properties that describe the types of events they generate or observe. These two basic types of ports are explained below:

a) **Service ports**: A service request is between a service requirement port and one or more service provision ports. There are two types of service ports:
   i) Discrete service port: It represents a stateless operation. It has a service descriptor property that describes the operation's signature.
   ii) Session service port: It represents a set of interdependent operations. It has an interface descriptor property that represents the operations the port handles. It also has state information on the status of the session service. Because the system builders can access component properties, there is a built-in flexibility in customizing the port's services.
When any part of the component needs services outside the component boundary, it sends a service request to one of the component's ports. Thus there is a contract between a component's internal parts and the ports in the component's boundary.

![Hierarchy of Ports](image)

**Figure 3.3: Hierarchy of Ports**

b) Event ports: The EventGeneratePort sends the event object to all the connected links. The EventObservePort receives the events.

A component with ports as its boundary is fully encapsulated and ready for assembly. It is concerned only with the component's core functionality and not on the dependencies outside the boundary. Because a port instance is simply a property of its component, it is easy to replace or customize. Links are used for intercomponent communication.

The basic function of a link is to deliver service requests or events to target ports. There are three types of links as shown in figure 3.4. They are:

a) **Service links**: They connect service request ports with service provision ports.
b) **Event links**: They connect event generation ports with event observation ports.

c) **Event-Service links**: They connect event generation ports with service provision ports.

Thus the three types of links correspond to service-service, event-event and event-service interaction styles. A link has two properties, namely, source ports and target ports. Links deliver service requests or event notifications received from source ports to target ports. A source port is either a service request port or an event generation port, depending on the link type. A target port must be either a service provision port or an event observation port depending on the link type. A port instance belongs to only one component. But a link instance belongs to multiple ports. Each type of network protocol requires a different link type. The proposed model implements CORBA/IIOP, RMI, RMI/IIOP and local links.
Ports and links are components that can be reusable, replaceable and customizable. The plug and play ability of links also supports system evolution. Replacing an IIOPSESSIONSERVICELINK by an RMISESSIONSERVICELINK, for example, will neither break the components being connected nor require that they are changed. The key issue is that the functionality of links is independent of the component core functionality.

A case study on reusability analysis is performed on e-shopping framework using banks, credit cards and e-cards for three different object technologies namely, RMI, RMI/IIOP and CORBA. Both object oriented approach and application approach have been used and tested. The following paragraphs explain the design of the experimental e-shopping framework.

3.3.3 Design of eshopping framework

Three different frameworks for eshopping were designed and implemented using object oriented approach and application oriented approach. Then a reuse analysis was performed in both these approaches. This section explains the design of the eshopping frameworks using banks, credit cards and ecards.

a) eshopping framework using banks:

The eshopping framework consists of Customer, Merchant, Middleman, Merchant Bank (M Bank) and Customer Bank (C Bank) components as shown in figure 3.5. The elements surrounding each component are ports. The interactions between components in this framework as labeled in figure 3.5 is listed below:

1) Customer requests for service from the merchant through Service Required port.
2) Merchant requests for service from the C Bank. C Bank verifies for validity of the amount and completes the service.
3) Merchant generates events to C Bank and M bank to update the amounts.
4) Merchant generates a completion event to the customer.
b) eShopping framework using Credit Cards

The interactions between components in this framework as labeled in figure 3.6 is explained below:

1) Customer requests for service from the merchant.
2) Merchant requests for service from the Middleman.
3) Middleman requests service from C Bank. C Bank verifies for validity of the amount and completes the service and returns the result to middleman. Middleman returns the result to Merchant.
4) Merchant generates events to C Bank and M bank to update the amounts.
5) Merchant generates a completion event to the customer.

Figure 3.5: eShopping framework using banks

Figure 3.6: eShopping framework using Credit Cards
In this framework, the customer, M Bank and C Bank components are reused. Merchant component’s service port is changed to provide a link to the middleman and middleman component is newly introduced, when compared to eshopping framework using banks.

c) eshopping framework using ecards:

The operation of this framework as shown in figure 3.7 is explained below:
1) Customer requests Middleman to generate ecard. Middleman transfers the ecard number to the customer.
2) Customer requests for service from the merchant using the ecard generated.
3) Merchant requests for service from the Middleman.
4) Middleman requests service from C Bank. C Bank verifies for validity of the amount and completes the service and returns the result to middleman. Middleman returns the result to Merchant.
5) Middleman generates events to C Bank to update the amount.
6) Merchant generates events to M Bank to update the amount.
7) Merchant generates a completion event to the customer.

Figure 3.7: eshopping framework using ecards.
In this framework, extra ports and links are provided between the customer and middleman components for ecard generation. The components Customer, M Bank, C Bank, Merchant and middleman are reused from the credit card module.

The experimental results of reuse analysis in using these frameworks in object-oriented approach and application-oriented approach is presented in this chapter.

3.4 Performance Analysis of Distributed Object Middleware

The most important criteria for selecting an appropriate distributed object model for a target application is its performance. Distributed object models are not as time efficient as low-level approaches. The main reasons for this include marshalling and demarshalling overhead, demultiplexing and dispatching overhead, data copying and additional remote invocations for resolving object references. The overhead of CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface is due to non-optimal internal buffer management strategies [143]. It is desirable to permit reuse of requests if the operations are oneway and the parameter values do not change. Performance results of benchmarking sockets and CORBA implementations over Ethernet and ATM networks have been studied. Results show that the latency for CORBA implementations [144] [145] is relatively high due to non-optimized internal buffering and presentation layer conversion overhead in CORBA implementation.

3.4.1 Performance Evaluation Metrics

This section describes the benchmarks to evaluate the performance of Java RMI, RMI-IIOP, CORBA and DCOM from a developer's standpoint. For a developer the most important performance criterion is the total time taken by a remote method invocation and throughput. This time is defined as the Round Trip Time (RTT). RTT is the time that elapses between the initiation of a method invocation by the client until the results are returned to the client. RTT has to be evaluated for different data types, sizes and for multiclient
scenarios. Throughput is defined as the number of method invocations in a given time interval. Throughput and RTT are inversely proportional.

Performance degradation when multiple clients simultaneously interact with a single server is defined as a function of number of simultaneous clients. Performance degradation as a base index is computed as quotient between the RTT by the given number of simultaneous clients and the RTT by a single client. Performance degradation as a chain index is computed as quotient between the RTT by the given number of simultaneous clients and the RTT by a given number of clients minus one.

### Table 3.1: Simple Data Types used for Performance Evaluation

<table>
<thead>
<tr>
<th>Java RMI data type</th>
<th>CORBA IDL data type</th>
<th>RMI-IIOP/IDL data type</th>
<th>COM IDL data type</th>
<th>Length in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>boolean</td>
<td>boolean</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>wchar</td>
<td>char</td>
<td>char/wchar_t</td>
<td>16/8</td>
</tr>
<tr>
<td>byte</td>
<td>octet</td>
<td>octet</td>
<td>byte</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>short</td>
<td>short,small</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>long</td>
<td>long</td>
<td>int/long</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>long long</td>
<td>long long</td>
<td>hyper</td>
<td>64</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
<td>float</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>double</td>
<td>double</td>
<td>64</td>
</tr>
</tbody>
</table>

RTT has been measured for different primitive data types shown in table 3.1. Table 3.1 presents the equivalent mapping between Java RMI and RMI-IIOP/IDL, CORBA IDL basic data types. The mapping of Java RMI boolean, short, int, long, float and double is straightforward because they have exact IDL analogous types. The Java RMI signed data type byte is mapped to the unsigned IDL type octet as both are 8 bits long. The Java RMI unicode char is mapped to the IDL wchar type.

Section 3.4 gives the experimental results on evaluating RMI, RMI/IIOP, CORBA and DCOM for different data types, sizes and for multiclient scenarios. This section also presents an overhead analysis for these models.
3.5 Experimental Results

This section presents the results of reuse and performance analysis of different distributed object models.

3.5.1 Reuse Analysis

The measures used in the analysis of the various components in the eshopping frameworks are listed below:

- **Halstead Volume**: It is represented as 
  \[(N1+N2) \log_2(n1+n2)\]
  , where \(n1\) and \(N1\) are the total number of operators and the number of operators used in the module, and \(n2\) and \(N2\) are the total number of operations and the number of operations used in the module. If too small then the reuse becomes impractical. If too large then the component is error prone.

- **Mc Cabe measure**: This measure of Cyclomatic complexity is evaluated from flow of information between components. It is given as \(e-n+2\), where \(e\) is the number of connections between the components and \(n\) is the number of components.

These measures are used to evaluate the various components in the framework. It is shown in table 3.2. From table 3.2, it can be seen that these measures are within the preferred values as specified by Heinmann [144].

<table>
<thead>
<tr>
<th>Reuse measure</th>
<th>Preferred value</th>
<th>Evaluated value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Object oriented</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BK</td>
</tr>
<tr>
<td>a) Halstead Volume</td>
<td>&lt;40</td>
<td>22</td>
</tr>
<tr>
<td>b) Mc Cabe Measure</td>
<td>&lt;10</td>
<td>3</td>
</tr>
<tr>
<td>c) average method size</td>
<td>&lt;24</td>
<td>15</td>
</tr>
<tr>
<td>d) average number</td>
<td>&lt;6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>of methods per</td>
<td></td>
</tr>
<tr>
<td></td>
<td>class</td>
<td></td>
</tr>
<tr>
<td>e) average number</td>
<td>&lt;6</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>of instance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>variables</td>
<td></td>
</tr>
<tr>
<td>f) depth of class</td>
<td>&lt;6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>hierarchy</td>
<td></td>
</tr>
<tr>
<td>g) average size of</td>
<td>&lt;200 SLOC</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>component</td>
<td></td>
</tr>
</tbody>
</table>

BK: Banking framework; CC: Credit Card framework; EC: Ecard framework
The various measures of reuse of components of eshopping frameworks include the following:

- **Reuse leverage** is concerned with the number of reused objects. It is \( \frac{\text{number of objects reused}}{\text{number of objects built}} \).

- **Quality of the system** is evaluated based on the number of reused lines of code. When components are reused, the quality of a system improves as the defect rate reduces. The defect rate of reused code is around 0.9 defects/KLOC and for newly developed components, it is around 4.1 defects/KLOC [144].

These reuse measures of components in two different models is tabulated in table 3.3.

**Table 3.3: Evaluation measures for Reuse**

<table>
<thead>
<tr>
<th>Reuse Measures</th>
<th>Object oriented approach</th>
<th>Application oriented approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Reuse percentage</td>
<td>83.2%</td>
<td>79.6%</td>
</tr>
<tr>
<td>Reuse leverage</td>
<td>.66</td>
<td>.4</td>
</tr>
<tr>
<td>Productivity</td>
<td>69.33%</td>
<td>66.1%</td>
</tr>
<tr>
<td>Quality improvement</td>
<td>65%</td>
<td>62%</td>
</tr>
</tbody>
</table>

A: Reuse of Banking framework in Credit Card framework  
B: Reuse of Banking framework in ecard framework  
C: Reuse of credit card framework in ecard framework.

The following components are used more than once:

1) Validation component of credit card system is reused in the e-cards transaction.
2) Component for amount checking is used in credit card and e-card transaction
3) Banking component is used in both the systems.
4) Merchant shopping cart is used in both the systems.

From table 3.3, it can be seen that even though objects from banking components are reused in ecard framework, since more number of new objects are built, its reuse leverage is less when compared to reuse leverage of banking components in Credit Card framework.
3.5.2 Performance Analysis

This section compares the performance of the four major distributed object models, namely, RMI, RMI-IIOP, CORBA and DCOM. The performance is measured in terms of RTT and throughput. To measure RTT, interfaces with corresponding classes have been defined on the server side for each basic data type listed in table 3.1 and for two compound user defined data types, namely, testStruct (transfer by value) and myObject (transfer by reference). testStruct is a structure composed of 6 elements, namely, boolean, octet, short, long, float and double as its elements. myObject is an object having a short attribute as its data element. Each interface provides two methods - one that accepts different data types as a parameter and has no return value, and one that has no parameters but returns the data type as a return value. In both methods the processing overhead has been omitted.

For all the performance measurements the Sun Java 2 SDK, Standard Edition, version 1.3 has been used. For the code analysis the Intuitive Systems Institution's Optimizelt Professional profiler has been used. All the computers used Microsoft Windows NT 4.0 Workstation with Service Pack 6 as their operating system. COM measurements have been taken using VJ++ ver 6.0. Pentium IV-2.7GHz computers with 512 MB RAM have been used. All the performance measurements have been repeated 20 times and the average values are reported. Additionally each method has been invoked 200 times to achieve the necessary timing accuracy. To measure RTT the following steps are followed:

- The client reads the system time before invocation,
- It performs the method invocations.
- It reads the system time again and calculates the RTT for one observation.
- After completing 20 observations, mean of RTT is calculated.

The performance analysis of different distributed object models is done for both for single and multiclient scenarios.
a) Single Client Scenarios

A comparison on the performance of primitive data types, Strings, testStruct and myObject for single client scenarios in RMI, RMI-IIOP, CORBA and COM is presented in figures 3.8 to 3.11. Figure 3.8 presents the comparison of the RTT for basic data types in CORBA, RMI-IIOP, RMI and COM. RMI-JRMP has achieved the best performance when compared to CORBA and RMI-IIOP. For 16K arrays RMI has a RTT for RMI is 2 msec, whereas for RMI-IIOP it is 4.2 msec and CORBA it is 5 msec. For 16 K arrays COM RTT takes 22 microsec (COM measurements are taken using VJ++).

![Figure 3.8: RTT Comparison for Primitive Data Types](image)

Figure 3.9 shows the RTT for strings. Both RMI and RMI-IIOP use unicode characters, whereas CORBA-IDL uses 8 bit characters. Hence CORBA performs well when compared to RMI-IIOP. It is found that COM has higher throughput when compared to all the three models. But, COM measurements have been taken using VJ++ ver 6.0. Figure 3.10 shows the RTT for testStruct. RMI and CORBA have comparable results, although RMI transfers testStruct as object by value and CORBA as a simple data structure. COM has higher throughput when compared to the other models.
Figure 3.9: RTT Comparison for Strings

Note: COM measurements in Figure 3.8, 3.9, 3.10 and 3.11 are plotted against the secondary Y axis

Figure 3.10: RTT Comparison for TestStruct

Figure 3.11: RTT Comparison for myObject
As shown in figure 3.11, RTT for myObject (object by value) is low for RMI-IIOP and RMI when compared to CORBA-IIOP. Again, it is noted that COM achieved the best performance, but the measurements were taken using VJ++ ver 5.0, whereas the other models were evaluated using J2SDK ver 1.3.

The comparative performance of RMI-IIOP and CORBA with respect to RMI is shown in table 3.4. In all cases except in the case of testStruct, RMI-JRMP shows the best performance. CORBA-IIOP and RMI-IIOP are slower. This is because RMI and RMI-IIOP pass structures as objects by value, whereas CORBA passes the elements as different parameters to the function call. In the case of primitive data types and its arrays, corba shows less performance when compared to rmi-iiop, whereas in the case of myobject (object by value) and strings, CORBA performs better when compared to RMI/IIOP. This low throughput in RMI/IIOP and CORBA when compared to RMI are due to inefficiency in marshalling procedures.

### b) Multiclient Scenarios

Performance levels offered by RMI, CORBA/IIOP and RMI/IIOP in multiclient scenarios, for one to eight clients is shown in figures 3.12 and 3.13. In RMI and RMI-IIOP there is a linear increase in RTT. For CORBA-IIOP the RTT rises faster for first four clients and slower for others. For 4 clients RMI is

<table>
<thead>
<tr>
<th>Data Type</th>
<th>% Performance improvement of RMI wrt RMI/IIOP</th>
<th>% Performance improvement of RMI wrt CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic data types</td>
<td>54%</td>
<td>57%</td>
</tr>
<tr>
<td>String</td>
<td>33%</td>
<td>17%</td>
</tr>
<tr>
<td>Struct</td>
<td>22%</td>
<td>-46% (regression)</td>
</tr>
<tr>
<td>MyObject</td>
<td>42%</td>
<td>13%</td>
</tr>
</tbody>
</table>

Note: % Performance improvement of RMI wrt RMI/IIOP = \( ((RMI \text{ throughput} - RMI/IIOP \text{ throughput})/RMI \text{ throughput}) \times 100 \)
80% faster and by 8 clients 25% faster than CORBA. RMI was also 70% faster by 4 clients and 50% faster by 8 clients than RMI/IIOP. When observing the performance degradation we see that all the three models are comparable. By 8 clients, CORBA's relative degradation is even slower than RMI's, although RMI is better for very smaller number of clients. RMI-IIOP degradation is a little higher than that for RMI.

![Figure 3.12: Average method invocations in Multiclient Scenarios](image)

Note: COM measurements in figure 3.12 are plotted against the secondary Y axis

![Figure 3.13: Performance Degradation in Multiclient Scenarios](image)
c) Overhead Analysis

i) CORBA / IIOP Overhead Analysis

To process the client requests, the IDL server object creates a pool of threads. The Java IDL listener thread handles the low level connection details. For each client a Java IDL reader thread is created which handles the connection with the specific client. For each client also a pool of Worker threads is created. These threads process the IIOP protocol input streams. In the Listener Thread 5% of the total time was spent, while in the Reader thread 80% and in the Worker thread 15% of the total time was spent.

On the server side, for the Listener Thread, a lot of time was spent in native methods, like socketAccept (28% of the total time) and getHostAddr (22% of the total time). Around 17% of the total time was spent in thread management. In the reader Thread, the majority of time was spent in reading the socket (56% to 59% of the total time) and in creating new Worker threads (25% to 49% of the total time).

In the Worker Thread, depending on the data size, there are a few time consuming methods. The method read_octet reads the subsequent octet into the buffer (1024 bytes). Before this happens, alignAndCheck method is used (for data alignment and buffer enlargement). The method write_octet writes the subsequent octet to the buffer. It uses the alignAndReserve method which supervises the buffer size and calls the grow method to enlarge it. The three methods take 42% of execution time. The TypeCodeImpl.copy copies the input stream to the output stream and takes 33% of the time.

On the IDL client side there are two important threads. The reader Thread spends majority of its time in java.net.SocketInputStream.socketRead method. In the main thread 90% of the time is spent. With increasing data size the majority of the time is spent on alignAndReserve, alignAndCheck, grow, TypeCodeImpl.copy, write_octet and read_octet (92% of the total time).
ii) RMI / IIOP Overhead Analysis

On the server side, RMI-IIOP uses the thread pool with Listener, Reader and Worker Threads. The behavior of the Listener and Worker threads is similar as in IDL. In the reader thread the majority of time is spent in the socket communication statement socketRead (50%) and in the thread creation (38%). With the increasing data size, the initialization of the IIOP input stream becomes time consuming.

RMI/IIOP client creates two threads, the Reader Thread which handles the low-level communication and the main thread. The majority of the time in the Reader Thread is spent in the socketRead method (88%). With the increasing data size the method to process input becomes time consuming (11%). The client side overhead for the main thread is comparable to the IDL client side main thread. For RMI-IIOP however the grow method has a much larger share of upto 26% of the total time, compared to 11% for IDL. Also more time is spent in writing to sockets.

Based on the overhead analysis, the following overheads have been identified:

- An important source of the receiver side overhead is in the demultiplexing, demarshalling and presentation layers. CORBA-IIOP and RMI-IIOP use inefficient algorithms for demultiplexing which perform heavy checking (alignAndCheck). They also use bad buffering algorithms, which requires buffer enlargements dynamically and lead to excessive data copying.

- An important source of the sender side overhead is inefficient multiplexing. This is partly due to inefficient algorithms which perform aligning and reserving (alignAndReserve), has a small initial buffer size and inefficient buffer growth method (grow).

- Vast amount of the communication overhead resides in the low-level methods that take care for socket based communication. This includes socketWrite, socketRead and socketAccept methods. These methods are not implemented in Java. Rather these methods are written natively for
each target platform in C language and use the underlying OS calls. To access them JNI is used.

- CORBA and RMI-IIOP use an inefficient algorithm for demarshalling of strings. It uses java.lang.String which is a static object.
- Other sources of overhead lie in excessive data copying and in the generation of stubs and skeletons. While multiplexing and demultiplexing floats and doubles, they are first converted to integers. Further source of inefficiency is in the creation of redundant arrays.

3.6 Summary

A reusability analysis has been carried out for three different distributed object models in object oriented approach and application oriented approach. The results show good reuse percentage, good reuse leverage, good productivity and reduction in defect rate due to the reuse of components of credit card system in the e-card system. Also application oriented assembly of components facilitates the dynamic reuse of the links and ports. This further increases the reuse percentage and productivity in the application oriented systems.

Socket programming can be used to invoke the remote objects. As it is low-level and time consuming for modern application development, RMI provides a transparent method of invocation between objects on different Java Virtual Machines (JVM). Since RMI is language dependent, CORBA is used in heterogeneous environment. RMI/IIOP facilitates remote method invocation over IIOP. A comparison on the performance of four important distributed object models, namely, RMI, RMI/IIOP, CORBA and DCOM has been carried out. Round Trip Time (RTT) has been measured for different data types as parameters and return values, different data sizes and for multi client interactions. RMI was noted to perform better than CORBA by 13% to 57% (except for Strings, where CORBA performs by 17% better than RMI) and RMI/IIOP by 22% to 54%. RMI uses JRMP as its communication protocol, where as CORBA uses IIOP. The overhead analysis performed using Optimiselt reveals that IIOP suffers from multiplexing, alignment and buffering
inefficiencies, excessive data copying, use of JNI for communication and deficits in marshalling.

Based on the overhead analysis done in CORBA, this thesis proposes certain optimizations to overcome the inefficiencies in CORBA-based applications. It proposes novel methods addressing the following issues to improve process of presentation conversion.

- Minimization of the alignment problems.
- Reduction in the size of stub code.
- Reduction in the number of copy operations.
- Improvements on the marshalling speed.
- Improvements on the communication protocol.