SCALABILITY ISSUES IN CORBA

P . Manjula Rani, A. Vijay Srinivas and D Janaki Ram
{pmrani, avs, janaki@lotus.iitm.ernet.in}
Distributed Object Systems Lab, Department of Computer Science & Engg.,
Indian Institute of Technology, Madras, India.

Abstract

Scalability is recognised as a primary factor to be considered in the design of distributed systems. The scalability of object oriented middleware CORBA becomes a major concern as it has emerged as a standard architecture for distributed object computing.In this paper, a systematic scalability analysis of the basic components of CORBA specification is attempted. From this analysis, Portable Object Adapter (POA) and Implementation Repository(IR) are identified to influence scale of the CORBA based system. The specification of POA provides enough flexibility for the application designer to handle scalability. The existing implementations of IR have a trade off between scalability and object migration. A scalable design of the IR is proposed which allows individual objects to migrate without compromising scalability. A performance comparison of the proposed model with the existing IR designs is made using a simulation study.

1 Introduction

Scalability is a major concern in the design of distributed architectures and applications [13]. Scale can refer to any one of the various input factors depending on the system considered. For example, in a distributed file system, it could be the number of nodes or users in the system [10]. Growth is an integral quality of a distributed system. A system that works well at smaller scales inherently fails as it grows if scale is not considered in the design.

A distributed system is said to be scalable, if the performance is within the tolerance limits as scale increases upto a maximum limit, based on the application [9]. For example, for a distributed network management system, the scalability limit could be millions of managed devices. But for a telephone system, it could be a billion customers. The performance degradation allowed as scale increases also depends on the type of application. There is no general specification that dictates the performance penalty that can be paid for scalability.

An important characteristic of current environments such as internet and intranets is that they are highly heterogeneous consisting of different hardware platforms and operating systems. Development of object-oriented software applications that make use of heterogeneous networked systems has led to the development of communication middleware like Common Object Request Broker Architecture (CORBA) [14], Distributed Component Object Model (DCOM) [3] and Java Remote Method Invocation (RMI) [7]. The scalability of CORBA depends upon how well scalability has been addressed in the specification and its implementation by the ORB vendor. In this work, the number of objects in the distributed system is considered to be the scale factor. CORBA ORBs must scale predictably and efficiently as the number of objects in the end systems and distributed system increases. The end to end response time or latency is considered to be the performance criteria as the number of objects in the system increases. In this work, the scalability of CORBA is studied and it is found that the following three different levels affect the scale: the implementation level, the POA level and the implementation repository level.

At the implementation level, scalability can be achieved by optimizing any or all of the following overheads: overhead of acquiring the object reference, marshaling and unmarshaling times and the demultiplexing overhead. The POA specification allows the application designer to design for scale. As the number of objects in the system increases, the application designer can use a hierarchical design consisting of servant managers, multiple POAs with different policies and a POA manager. Existing designs of the IR can store entries either at the POA level in which case migration of individual objects is not possible, or at the individual object level which compromises scalability.

This paper proposes a scalable design of the IR

which has a hierarchical structure. A new way of locating migrated objects using message filters [8] at the POA level is also proposed. This allows individual objects to be migrated without involving the IR. Further, the IR can store entries at the POA level, which improves scalability. The performance of this model is compared with the existing IR designs using a simulation model.

The rest of the paper is organised as follows. Section 2 discusses the main scalability issues in CORBA. The proposed scalable IR is explained in section 3. The idea of locating migrated objects is described in section 4. Section 5 gives the comparison of the proposed design with existing IR designs. Section 6 discusses the related works on scalability. Section 7 gives future research directions and concludes the paper.

2 Scalability of CORBA

This work has identified the main components that influence the scalability of CORBA to be the POA and the IR. The optimizations at the implementation level that can improve the performance and hence scalability are discussed in this section. This section further describes the scalability analysis of CORBA at the POA level and the IR level.

2.1 Implementation level

The scalability of CORBA-based distributed system depends upon the performance overheads incurred in the invocation path [1]. In the client server scenario, the end to end latency is dependent on the sender side overhead, receiver side overhead and network overheads. The latency is also dependent on the type and size of data sent. As the number of servants increases, the demultiplexing overhead incurred determines the scalability of the system. Demultiplexing overhead is the time taken by the object adapter to locate the particular servant and the corresponding operation. The type of demultiplexing strategy used by an ORB affects the scalability of the system. The ORB may use any of linear search, perfect hashing or active demultiplexing for locating the servant. Linear search incurs a large overhead and is not scalable while the other two methods are more scalable.

To measure the CORBA scalability, the latency is measured with increasing number of objects in the end system server process. For these experiments, the public domain ORB *mico* [2] is used. The average latency is measured between two nodes for 100 client requests by increasing the number of servants in steps of 100, 200, 300 ... The number of requests per servant is restricted to 100. The servants are instantiated statically and client requests are generated in a linear order for all the servants. The total time taken is averaged over the number of servants instantiated. For mico, it is observed that the average latency remains unaffected as the number of servants increases upto 50,000 objects. Beyond that the performance starts degrading. This could be due to memory limitations in the machine.

2.2 Portable Object Adapter (POA) Level

The POA links the CORBA objects to the programming language servants. POA is responsible for creating Interoperable Object References (IOR), activating objects and dispatching requests made on the objects to their respective servants. The components of the POA architecture include POA client, POA server, POA manager, POA, servant manager and POA servant. The POA server is a process which hosts an implementation of the Interface Definition Language (IDL) interface. The servant implements this interface in the programming language which is the mapping for the given IDL interface. It implements the operations defined in this interface. The incoming requests from the client into the POA is controlled by a POA manager object associated with the POA. The POA manager controls the life cycle of the POA. It is used to allow requests to pass to the POA unimpeded, to discard or hold requests or to deactivate all request handling. POA maintains an active object map which maintains the mapping between the object id and the servant pointers. Each POA is associated with servants which are governed by a set of policies. More than one POA can exist in a server process, each governed by a different set of policies. These policies provide a mechanism for specifying various options for the management of objects.

Based on the application and requirements, a POA can

- use distinct servants for each transient CORBA object.
- only one servant to incarnate many persistent CORBA objects.
- use servant managers when an application containing many thousands of objects may want to incarnate only those objects that actually receive requests.
- a combination of all these
- use a default servant that carries out all requests.

If the POA policy needs to be different for different objects, multiple POAs can be defined. The application controls the capabilities of each POA, by assigning policies to each POA at creation time. From the scalability point of view, an application has the maximum flexibility to choose between the various configurations and POA policies to suit the requirements. For large applications, the POA can assign servant managers to handle the objects. This leads to increase in the number of levels since a client request has to traverse POA manager, POA, servant manager, servants and finally the operations. This layered architecture of POA may affect scalability since the overhead incurred increases. Also, as the number of objects in the server process increases, the ability of POA manager to handle the rate of requests coming in may affect the scalability.

2.3 Implementation Repository (IR) Level

Hosts which are configured to use the same IR are said to be in a single location domain. The existing designs of IR use either a small location domain or a large location domain [11]. In this paper, these two designs are named as SLD model and LLD model respectively.

2.3.1 **SLD Model** The location domain is said to be small if only a few nodes use a single IR. For our study, every node in the system is considered to be associated with a separate IR which runs on the same node. It is responsible for servers created in that node. In this case, the number of servers to be handled by this IR is smaller and results in high performance. The communication between the IR and the server does not involve the network. Failure of a node affects only those objects created in this node as they cannot be accessed. None of the other servers are affected due to individual node failure.

The drawback of this approach is the difficulty in migrating objects across location domains. An object, that migrates from one node to another, registers the new location with the old IR as well as the new IR. It results in a number of location forwards for the client requests if objects move further. Any client already bound contacts the old IR to locate the object. This results in a location forward to the new IR. As the scale increases, the number of objects residing in each node increases and the load on each node also increases. This in turn results in increased number of object migrations[1] and increased number of location forwards. Hence the time taken to find a migrated object increases at large scale.

2.3.2 **LLD Model** Many nodes are configured to use the same IR through the network in the large location domain. The IORs of all the objects created in this domain contain the address of the same IR in the object key. The clients need only connect to the IR. When an object moves, it notifies the IR, reregisters with it with the new host address and port number. This results in ease of object migration within the repository's location domain. As the number of objects increase, the server table becomes very large. This results in increased overhead due to lookup. The number of servers registering with the IR also increases which affects the performance. The IR has to handle client requests, server registrations and notifications of migrated objects.

The choice between these two designs depends on the trade-off between object migration and size of the server table at large scales. The IR can store entries either at individual object level or at POA level. Migration of individual objects requires storing the individual object locations. At larger scales, storing at individual object level results in large server table which leads to increase in lookup overheads. Individual object migration will not be possible if IR stores at POA level since POA name is used as a key to index the server table. If an object moves, all objects within the POA need to move with it. By having multiple POAs, the number of objects in a single POA can be reduced but the number of POAs in a process should be small for maintainability. Therefore the performance of these designs degrade at large scales.

2.4 Other Optimizations

As the number of client requests for the same server increases, multithreading the server will improve performance and hence scalability. For even larger applications, multithreading may not be adequate and a single server may be insufficient to handle the client requests. To improve scalability and performance further, a federation of servers can be used [12].

This study focuses on the basic ORB and server side optimizations. The scalability can also be improved by using client side optimizations. The object references can be cached by the client for future references to the server. If the request is for a server created in the same ORB, the object key can be decoded by the client to get the POA name of the server. If the client had referenced an object within this POA earlier, it knows the location of the server. Hence it does not have to go through the IR. Asynchronous invocation of client requests helps in improving the performance in large scale systems. This can be achieved using Asynchronous Method Invocation (AMI) [15], currently a separate messaging specification. The drafting of AMI into the CORBA specification and its subsequent implementation by

[1] One of the reasons for migrating objects is load balancing.

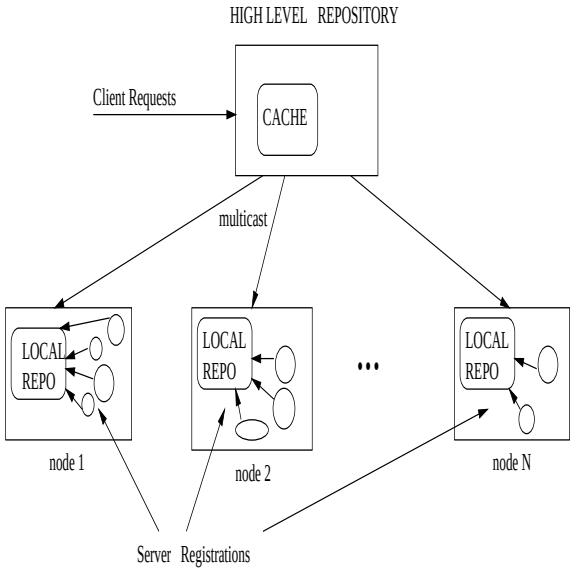

Figure 1: An Illustration of the model

```
interface high_level_repository {
        Object locate_server(...);
        Object activate_server(...);
        // These methods are used by
        clients to locate and activate
        servers.
}

interface local_repository {
        void register_server(...);
        void reregister_server(...);
        void remove_server(...);
        Object activate_server(...);
        //These methods are used by
        servers to register, activate
        or delete server information.
}
```

Figure 2: Methods in the IR Interface

the vendors become very important.

3 Proposed Model

In the existing ORBs, the LLD model and the SLD model are two ways of designing the implementation repository, as explained in the previous section. These two models have the tradeoff between scalability and object migration. Hence, a new way of designing a scalable implementation repository is proposed in this work.

The proposed Implementation Repository consists of a two level hierarchical structure as shown in figure 1, with two kinds of repositories. Each machine has a /it local repository. Servers created in each machine register with its own local repository. A group of such machines are configured to use a *high level repository*. These group of machines form a *single location domain*. The IOR is configured to hold the address of the high level repository. All client requests for servers within this domain are handled by the high level repository. The high level repository maintains a cache of the entries in the local repositories. When a client request arrives, the high level repository looks up in the cache and returns the server location. If the entry is not present, it sends a multicast request to the local repositories within its domain. One of the local repositories returns the location which is cached and returned to the client.

The binding algorithm, when a client invokes a request, works as follows:

- The client contacts the high level repository as the IOR contains its address.
- If the server entry is present in the high level repository and the server is active, it returns the address of the server in a location forward reply.
- If the server entry is present in the high level repository and the server is not active, the high level repository activates the server and returns the address of the server.
- If the entry is not present, the high level repository issues a multicast to all the local repositories. The reply is cached and returned to the client. If there is no reply which implies the server is not registered, it times out and returns an OBJECT_NOT_EXIST exception.

In this model, a major improvement in performance and hence scalability is achieved since the server registrations and client requests are handled by different repositories. All the entries in server table are referenced using POA name. Therefore, the size of the server table is manageable at large scales. Location of the migrated objects are handled at the POA level[2]. Hence, it does not involve the repository. This implies that the high level repository can handle larger client request rates at higher scales. The performance further improves based on the caching algorithm used since it reduces the access to local repositories.

Another key advantage over existing ORBs is that it facilitates POA level entries and individual object

[2]The method of locating migrated objects is explained in section 4.

migrations. In the existing ORBs, if entries are at POA level, individual objects cannot migrate. Storing at individual object level results in poor scalability. Fault tolerance with respect to IR is another major advantage over existing designs. If the high level repository fails, the cache can be reloaded from the local IRs. Whereas in the LLD model, if the IR fails, clients can no longer connect to objects within that domain. In the case of SLD model, IR failure is similar to failure of local repository in the hierarchical model.

The implementation repository therefore has two interfaces - one for the high level repository for binding the clients and the other for the local repository for administrative commands like registering the server etc. Figure 2 shows these two interfaces. The existing designs combine these operations in a single interface. The design of two different types of repositories can be seen as the only drawback of the proposed design. But the advantages it provides at large scales outweighs this difficulty.

4 Locating Migrated Objects

Object migration is important for a variety of reasons such as load balancing, fault tolerance, availability, performance etc. The concept of object migration and location of the migrated objects within a limited number of nodes is discussed in [5] for the *Emerald* system. In this design, all the nodes traversed by the object keep track of the new location and a time stamp to indicate its validity. There is no concept of a repository in this approach. At large scale, it results in too many location forwards. Hence, the performance could degrade drastically.

In CORBA, the location of an object is maintained by the IR. Therefore, the existing designs keep track of objects by updating the entries in the repository. The migrating entity has the responsibility to update the entry in the repository. The repository can store entries either at POA level or at individual object level. This design restricts the granularity of object migration to the extreme design choices of migrating all the objects within the POA or migrating individual objects.

This work concerns with locating a migrated object. It is assumed that the object is migrated by an external entity such as the client or a migration service. To cater to the scalability requirements, the hierarchical model stores the repository entries at POA level. Therefore, individual object migration is not possible using the repository. In the proposed method, the concept of message filters [8] is used to handle individual object migration at POA level.

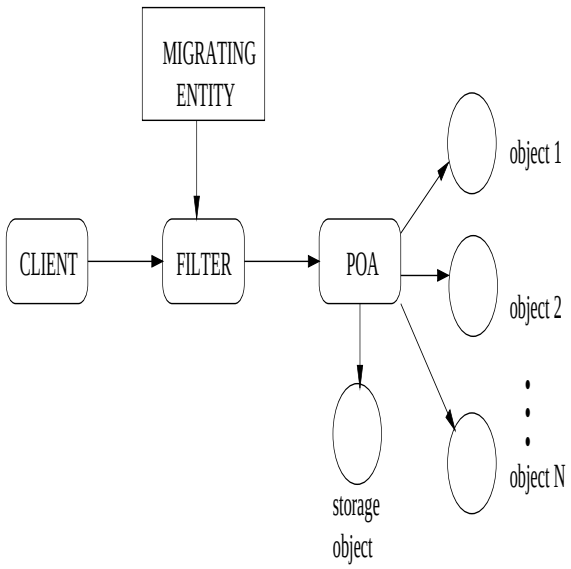

Figure 3: Object location mechanism

In the message filters paradigm, a language level relationship called *filter relationship* between the filter object (object which performs the filtering function) and the filter client (the object to be filtered) is introduced. Thus a filter object can intercept and manipulate messages sent to the filter client via filter member functions. The model supports both upward and downward filtering of messages. This means that, in the context of client-server computing, method invocation on server objects can be filtered transparent to the client by using a server side filter. The return values can also be filtered transparent to the server by using a client side filter.

Each CORBA object is associated with a POA as explained in previous sections. Therefore, the POA handles a group of such objects governed by the set of policies defined for this POA. The POA maintains an active object map which gives the mapping between the objectId and the object's servant location. A filter object is defined for the POA which intercepts all the methods that are defined for this POA. A separate storage object, specific to this POA, is defined to store the mapping between the objectId and the location of migrated objects as shown in figure 3 It also stores the startup command of the server. POA has to activate the object using the startup command if server is not already running. It contains a method to update these entries whenever an object within this POA is migrated by an external entity. When an *update* method is called on the object by the migrating entity, the filter redirects the request to the storage object. The storage object updates its entry with this new location. A method to *lookup* the new location

when a request arrives is also included in the storage object. The filter calls this method to find the new location of the migrated object.

The client generates requests using the persistent object reference which contain the IR address. Since IR stores the location at POA level, it returns the address of POA in a LOCATION_FORWARD reply. The client stub redirects the request to the POA which is intercepted by the filter. The filter checks if the request is for a migrated object by calling the lookup method in the storage object. The method is invoked normally using the active object map if the object is not migrated. For the migrated objects, the new location is returned to the client in a LOCATION_FORWARD reply.

The key idea is to use an object within the POA as a location broker for all objects within that POA. Every POA in the server application also has a filter which takes care of returning the new location of migrated objects. The reason for separating the storage and filtering functionalities into different objects is for extensibility. For instance, the filter can also be realised using ORB level interceptors. Both the functionalities cannot be handled by the interceptor as it requires the interceptor to be persistent.

5 Comparison of Proposed Model with Existing Models

The design of the proposed implementation repository is modeled using a discrete event simulation package called simjava [6]. The SLD model as well as LLD model are also modeled in a similar way for comparison.

The parameters considered for comparison are as follows:

- Client request interval, server registration interval and object migration interval are modeled as negative exponential functions with the given mean value.
- Hit ratio indicates the probability that the entry is found in the cache at the high level repository.
- POA active probability indicates the probability that the server is active at the time of client requests.
- Migration probability is the probability that the request is for a migrated object.
- Lookup time is the time taken for finding the entry in the repository
- Server activation delay is the time taken to activate the server.
- Migration delay is the time taken for locating the migrated object.
- Registration time refers to interval between server registrations with the local repositories.

The probability factors and the delays due to each of these factors vary with scale. The variation introduced is consistent across the models. The values given for POA active probability at the local machines, lookup time in the repository and server activation times are common amongst the three models. The variation in object migration delay is greater in the case of small domain since the number of location forwards increase with scale. The decrease in client request interval is also more since it includes requests generated due to migrated objects. The initial value of client request interval for SLD is larger since it handles objects in a single machine.

5.1 Simulation

The goal of the simulation study is to compare the relative performance of the hierarchical model when compared to the LLD and SLD models. The response time taken by the implementation repository to service the client requests is observed with increase in the number of entries in the repository. The number of entries reflect the number of objects in the system. The simulation gives the saturation point where the response time starts degrading with increase in number of entries. The model specific parameters are chosen to bring out the relative difference between the designs. For each simulation run, the number of entries are increased and the parameters are varied accordingly. Hierarchical model stores server registrations at the POA level. Therefore the entries in the repository are chosen to be at POA level in both LLD model and SLD model as well. It is assumed that an entry in the repository refers to an average of 200 objects present in the system[3].

5.1.1 **Comparison of Hierarchical and LLD Models** The number of nodes in the system for both the designs are assumed to be the same. The nodes containing the IR in LLD model and higher level repository in hierarchical model are considered to be dedicated machines. The only load arising on these nodes is due to the requests received by the repositories.

Since the number of objects in the system considered are same, the mean client request interval is the

[3] Under the assumption that a POA hosts 200 objects on an average.

same for both models. This interval reduces linearly with increase in number of entries and the variation is same for both models.

The server registrations in LLD model is with the same repository with a low errequest interval since it caters to all the machines in the system considered. In the case of the proposed model the server registrations are with local repositories and hence, the request interval is larger for the same number of servers. The mean server registration interval time remains a constant with scale since it is assumed that new servers register with the repositories at regular intervals.

Hit probability is associated with only the hierarchical model. Its value depends on the caching algorithm chosen such as LRU policy. The value c hosen is optimistic assuming a reasonable performance. This value decreases linearly with scale. The performance of the proposed scheme with varying hit ratios is presented in figure 4. It can be seen that the performance deteriorates as the hit ratio decreases.

The probability that the server is active is same for the local repository in the hierarchical model and the repository in the LLD model. Its value is higher for the high level repository since recently accessed entries are expected to be cached in the server table. This value decreases linearly with scale.

The lookup dela yis the same for the high level repository in the hierarc hical modeland the repository in the LLD model. The local lookup delay is applicable only to the hierarc hical model. The lookup dela ysincrease linearly with scale since the load on the system increases.

Migration interval is chosen to be higher when compared to the server registration intervalin the LLD model. It decreases linearly with scale. It is not applicable in the hierarchical model as the migration does not involve the repository.

Migration probability is applicable only in the hierarc hical model. It starts with a lo w value and increases linearly with scale. In the case of LLD model, the migrated object registers the new location with the repository. Hence, the repository always knows the correct location of the object. So the performance penalty is the same as for other objects. Whereas in the case of the proposedmodel, migration does not involve the high level repository. But locating a migrated object will take longer. Hence, the migration probability addresses the difference in location times of migrated objects and static objects.

The averageresponse time is measured for eac h simulation run consisting of 10000 iterations. Figure 5 shows the variation in response time with increase in the number of entries in the repositories. It can be observed that the hierarchical model performs better.

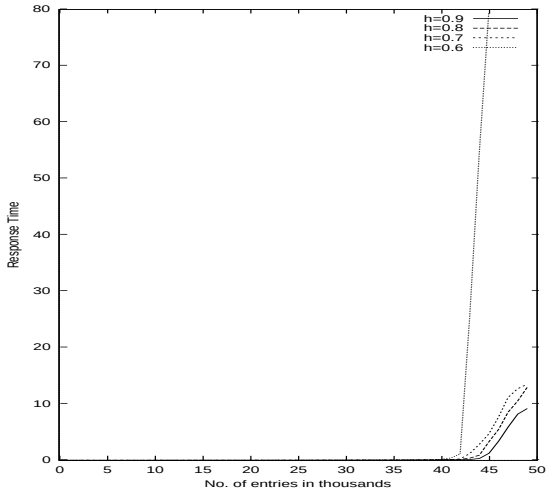

Figure 4: Comparison with different hit ratios

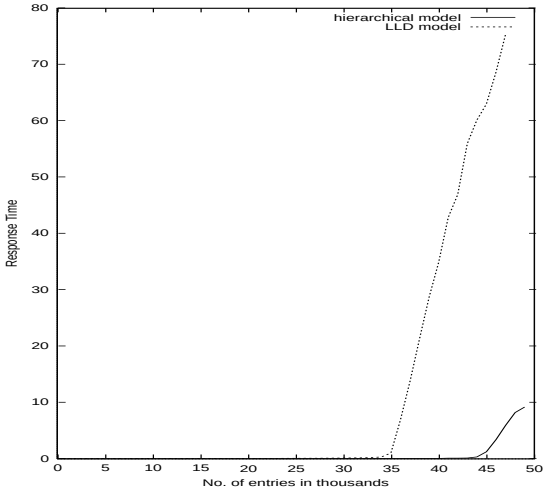

Figure 5: Comparison of Hierarchical and LLD models

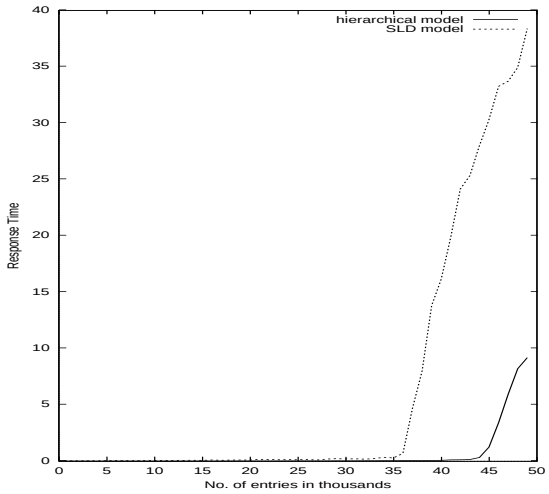

Figure 6: Comparison of Hierarchical and SLD models

The LLD model performs well upto 35000 entries and then degrades drastically. This is due to the fact that the single repository has to handle client requests, server registrations and migrating objects. To allow individual objects to migrate, LLD model has to store entries at object level which compromises scalability further.

5.1.2 **Comparison of Hierarchical and SLD Models** The simulation of the hierarchical model is same as explained in the previous section. In the SLD model, the IR is present in each node. Hence the client arrival interval in each node in the SLD model is chosen to be larger compared to hierarchical model where a single high level repository handles all the entries. The decrease in the arrival interval with scale is higher in LLD model since the client requests include the requests due to migrating objects. Migrating objects would leave the location forwards in IRs of all the nodes through which they migrate.

The server registration interval is same in both models since registration is done at each individual node. It remains constant with scale as in the previous case. Migration probability is also identical in both models. But the penalty paid to locate the migrated object is very large in SLD model. Migration interval for SLD model is same as in LLD model.

Figure 6 shows the comparison between the average response times observed with increase in the number of entries. At lower scales, the average response time is observed to be higher in the SLD model. This is explained by the fact that the IR in SLD model is on a loaded machine whereas the IR in the hierarchical model is on an unloaded machine. The average response time of the SLD model increases drastically beyond 35000 entries. This is due to increased number of migrated objects in the system. Here again, if the entries in the repository are at individual object level, the scalability would be affected drastically.

From the above observations, it can be seen that the rate of client requests determines the scalability point[4]. The system can be made to scale further by multithreading the IR. However, in the LLD model, the server registrations and the limit on the size of the server table along with the client request rate restrict the scalability point. In the SLD model, the scalability point gets restricted by the increased rate of migrating objects in addition to the client request rate. The hierarchical model is restricted only by the client request rates and hence, multithreading improves its performance drastically.

6 Related Work

The scalability of CORBA is discussed in [1]. [1] focuses on the implementation of the specification whereas this paper makes a comprehensive study of scalability. Further, according to [1], performance is the key to scalability. Whereas in this paper, performance is only one aspect of scalability of CORBA. [11] discusses scalability with respect to binding of persistent objects using the implementation repository. It discusses the pros and cons of the existing implementation repository designs and their effect on scalability due to the granularity of object migration. It suggests that the choice of the designs is based on the trade off between scalability and individual object migration. But this paper proposes a scalable repository design as well as a mechanism to allow individual objects to migrate. [4] is a comparison of three different client-agent-server interaction architectures over CORBA to test their scalability. It does not discuss the scalability of CORBA itself, rather it is more of a comparison of the three architectures from a scale perspective. [9] discusses the design of scalable distributed applications from the perspective of software engineering. According to [9], building scalable solutions depends on the application and general solutions such as replicating and distributing data does not make much sense.

7 Conclusion

A comprehensive study of scalability of the object oriented middleware CORBA is done. From this study POA and IR are identified to influence scalability of CORBA. A scalable design of IR has been proposed. Scalability is achieved by having a hierarchical

[4] Scalability point is the value of the scale factor beyond which the performance of the system changes drastically [16].

structure and separating the client requests handling, server registrations and location of migrated objects. A way of locating migrated objects using message filters is also proposed.

This paper has focused on the basic components of CORBA. The scalability can also be viewed from client side optimizations, federating servers and application design. Garbage collection of unused objects and dangling references in the system influences scalability from the perspective of memory usage. The problem of garbage collection is difficult to tackle and requires further research. Currently, garbage collection is left to the application to handle and there is no such provision at system level. The idea of locating the migrated object can be extended further to build a separate object migration service. This service can also include methods for actually migrating objects.

The study of scalability of CORBA is a step in understanding the design of scalable distributed object systems. Further research is required for generalizing the lessons learnt in this work and to be able to design scalable distributed systems. For instance, the proposed IR design can be extended as a scalable solution to the generic location problem in distributed systems.

The scalability of CORBA may very well determine its survival. Further, since CORBA is becoming a widely used middleware for distributed object computing, the scalability of CORBA could be a pointer to the scalability of a large class of distributed systems.

References

[1] Aniruddha S. Gokhale, Douglas C. Schmidt, "Measuring and optimizing CORBA latency and scalability over high speed networks", *IEEE Transactions on Computers*, Vol. 47, No. 4, April 1998.

[2] Arno Puder and Kay Romer, "MICO is CORBA", http://www.dpunkt.de/mico/.

[3] Distributed Component Object Model Protocol *DCOM/1.0* *http://msdn.microsoft.com/library/specs/distributedcomponentobjectmodelprotocol-com10.htm*

[4] Istabrak Abdul Fatah and Shikharesh Majumdar, "Performance Comparison of Architectures for client-server interactions in CORBA", http://www.sce.carleton.ca/faculty/majumdar/majumdar_pub.shtml.

[5] Eric Jul, Henry Levy, Norman Hutchinson and Andrew Black, "Fine-grained mobility in the Emerald system", *ACM Transactions on Computer Systems*, Vol. 6, No.1, Feb 1988.

[6] Fred Howell and Ross McNab, "simjava: A discrete event simulation package for Java, With applications in computer systems modelling", http://www.dcs.ed.ac.uk/home/hase/simjava/.

[7] Java Remote Method Invocation (RMI) Specification JDK 1.2, 1997-1998 Sun Microsystems Inc., http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[8] R. K. Joshi, Vivekananda N. and D. Janaki Ram, "Message Filters for Object Oriented Systems", *Software - Practice and Experience*, Vol. 27, No. 6, June 1997, pp. 677-699.

[9] Maarteen VanSteen, Stefen van der Zijden and Henk J. Sips, "Software Engineering for scalable distributed application", *Proceedings 22nd International Computer Software and Applications Conference (CompSac)*, Vienna, August 1998.

[10] Mahadev Satyanarayanan, "The influence of scale on distributed file system design", *IEEE Transactions on Software Engineering*, Vol. 18, No. 1,January 1992.

[11] Michi Henning, "Binding, Migration and Scalability in CORBA ", *Communications of the ACM*, Vol. 41, No. 10, October 1998.

[12] Michi Henning and Steve Vinoski, *Advanced CORBA Programming with C++*, Addison Wesley, Massachusetts, 1999.

[13] B. Neuman, "Scale in Distributed Systems", *Readings in Distributed Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 463-489.

[14] Object Management Group, "The Common Request Broker: Architecture and Specification", 2. 3. 1, October 1999.

[15] Object Management Group, "CORBA Messaging Specification", OMG Document orbos/98-05-05 ed., May 1998.

[16] Svend Frolund and Pankaj Garg, "Design-Time Simulation of a Large-Scale, Distributed Object System", *ACM Transactions on Modelling and Simulation*, Vol.8, No. 4, October 1998, pp. 374-400.