Performance of Parallel Architectures for CORBA-Based Systems

Ming Huo
Department of System and Computer Engineering
Carleton University
1125 Colonel by Drive, Ottawa Canada, K1S 5B6

Shikharesh Majumdar
Department of System and Computer Engineering
Carleton University
1125 Colonel by Drive, Ottawa Canada, K1S 5B6
majumdar@sce.carleton.ca

ABSTRACT
This research is concerned with achieving high performance on middleware-based inter-operable distributed object computing systems. This paper reports a preliminary investigation of the impact of using parallel architectures that use concurrent server invocations to improve performance. One of these architectures is observed to lead to a substantial performance improvement in comparison to the conventional sequential interaction architecture.

Keywords
CORBA performance, high performance middleware, interaction architectures.

1. INTRODUCTION
Distributed Object Computing (DOC) is a very popular paradigm for system implementation; it combines the attractive features of distributed computing such as reliability and concurrency with the well-known reusability properties of Object Oriented (OO) systems. Heterogeneity is natural in DOC systems. Different system components are often written in different programming languages and run on different operating systems. *Middleware* provides inter-operability in such heterogeneous DOC systems and enables a client written in a particular programming language running on top of a specific platform to communicate with a server implemented using a different programming language and platform. *Common Object Request Broker Architecture (CORBA)* is a middleware standard proposed by the Object Management Group (OMG) [7]. A number of Commercial-Off-The Shelf (COTS) middleware products that conform to the general CORBA standard are available. Although middleware provides inter-operability, if not designed carefully, a middleware-based system can incur a severe performance penalty. Engineering high performance is important in the context of a variety of different systems. High scalability and low latency are crucial in many applications that include telecommunication products, process control systems, and other performance demanding applications. This paper addresses these performance issues in CORBA-based systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSP'04 January 14-16, 2004, Redwood City, California.
Copyright 2004 ACM 1-58113-673-0/04/0001 ...$5.00.

Both the clients and servers in a CORBA-based system use a common standard Interface Definition Language (IDL) for interfacing with the Object Request Broker (ORB) that provides client-server inter-communication as well as a number of other facilities such as location and trading services through the ORB agent. Before invoking a method in a server object, the client binds with the object first. The binding request sent by the client is mapped by the ORB agent to a *handle* that is subsequently used by the client to contact the server.

A large body of research exists in the field of client-server systems. A representative set of previous work that has focused on CORBA is discussed. A detailed survey is available in [4]. A general description of the CORBA in terms of its concepts, roles and behaviors as well as its use in enterprise computing are presented in [9]. Its usage in Network Management System is discussed in [3]. Extending CORBA to real-time systems is also receiving attention from researchers [8]. High performance computing has concerned the concurrent processing of different parts of large data sets. Using CORBA for a parallel application for WZ matrix factorization is reported in [2]. A specification for "Data Parallel CORBA" is being developed by OMG for parallel processing applications [6]. The parallel architectures described in this paper, however, are primarily useful for exploiting *control parallelism.*

Many performance demanding systems that include telecommunication and embedded systems do not need absolute guarantees for meeting deadlines, but low latency and high scalability are highly desirable attributes of these systems. This research focuses on such performance demanding systems based on the general CORBA specifications. Our previous work on CORBA performance has demonstrated three different ways of performance optimization: guidelines for application design [10], effective client-middleware-server interaction architecture [1] and techniques that exploit limited heterogeneity in systems [11]. Since this research focuses on the second approach, a more detailed discussion of the client-middleware server interaction architectures is presented in the following section.

The relationship between software architecture and performance is the subject of attention for system designers and users. This paper presents a preliminary analysis of a number of parallel

client-agent-server interaction architectures that use concurrency in server execution for improving performance. The performances of these architectures are compared with that of the traditional sequential architecture. Exploitation of parallelism at the hardware and operating system level is well known. To the best of our knowledge there is no existing research that has analyzed the impact of parallel interaction architectures for CORBA middleware-based systems on performance. Using performance prototypes based on Iona's E2A ASP version 5.1 middleware [5] we have analyzed the performances of the parallel architectures deployed over a number of Linux PC's. The results demonstrate the superiority of some of the parallel interaction architectures especially at low to medium load.

The rest of the paper is organized as follows. The following section presents the parallel interaction architectures. The experimental environment used for their performance comparison is presented in Section 3. The following section describes the results of the experiments. Our conclusions are presented in Section 5.

2. INTERACTION ARCHITECTURES

Client-middleware-server interaction architectures can have a profound effect on system performance [1]. In this paper we have considered three different types of parallel architectures and a sequential architecture that are presented next. The synchronous parallel interaction architecture described in Section 2.3 corresponds to an architecture described in [1]. The other two parallel architectures are introduced in this paper.

Consider a system with multiple clients and four servers A, B, C, and D. A client runs cyclically and needs multiple server invocations to complete a job in each cycle. Each client calls two of these servers in the first cycle (Server A and one to Server B for example) and the other two servers in the second. This set of operations is repeated: the first set of servers is called in the third cycle followed by calls to the second set in the fourth cycle and so on.

Note that in the conventional systems the two servers are invoked sequentially in a cycle. A method in A is invoked first. When the client receives the results of the operation it invokes a method in B. When a request is made the client remains blocked until the result of the operation arrives from the server. This conventional interaction architecture is referred to as a *Sequential* architecture (S) in this paper. Note that 12 messages are interchanged in a client cycle with this architecture. For each server interaction 6 messages are required: 2 for requesting and receiving the handle of the desired server object from the ORB agent, 2 for handle verification (sending of the verification message by client and the acknowledgment from the server), and 2 for invoking the method in the server and receiving the results.

In many situations the operations performed by the two servers, Server A and Server B for example, are independent of each other and can be performed concurrently. This research focuses on using such parallel server invocations to improve system performance. Three different interaction architectures are proposed and their performances are evaluated. Each of these architectures uses a CORBA daemon that communicates with the client. The three architectures called the Handle Driven PP-Daemon-based architecture (HP), the Forwarding PP-Daemon-based architecture (FP) and the Synchronous PP-Daemon-based architecture (SP) are described in the following paragraphs. The main difference between the Sequential and these PP-daemon-based architectures is that the ORB agent is replaced by a PP-daemon that often provides additional functionality in comparison to the ORB agent that simply performs a name to handle mapping. Ideally a single program should be used to implement the PP-daemon. Since the source code of the middleware product E2 ASP is not available the PP-daemon is implemented as a process that is distinct from then ORB agent. Upon startup the PP-daemon obtains the handles of all the servers and renews them in a fixed interval. Note that this operation is not required in a system built from scratch in which the PP-daemon performs the duties of the ORB agent.

The Handle Driven PP-Daemon-based Architecture:
This architecture is similar to the sequential architecture in which the PP-daemon replaces the ORB agent. At the beginning of each cycle the tagged client sends a single request asking for the handles of Server A and Server B from the PP-daemon. After receiving the handles, the client uses the handles to invoke the servers concurrently. The client is multi-threaded: each thread uses a separate synchronous CORBA method invocation. The multithreading used in the client increases the code complexity slightly. By using an additional thread, the client can do other work while waiting for the results to arrive. After the client gets all the results, it will process them and obtain the result for the complete job.

This architecture leads to 10 messages in a client cycle. 2 messages are interchanged between the client and the PP-daemon: the client requesting all the server handles required for a job and the PP-daemon sending the handles back to the client. As in the case of the Sequential architecture 4 messages are required for the invocation of each server, leading to a total of 8 messages being exchanged between the client and the servers.

The Forwarding PP-Daemon-based Architecture:
With this architecture, the client sends the entire job consisting of multiple requests to servers to the PP-daemon. The requests are forwarded by the PP-daemon to the appropriate servers that execute concurrently. Each of the servers sends the result of the method invocation directly to the client. The client processes these results and obtains the final result corresponding to the entire job.

Except for the returning of results, all the interaction between clients, the PP-daemon and servers are asynchronous without any return value and only IP level acknowledgements are returned. The stringified client handle is sent along with the request to the PP-daemon and gets forwarded to the servers. The servers use this client handle to send the results back to the client. The client can do other work while waiting for the results to arrive. Each server calls a designated method in the client to return the results generated by the method invoked by the client. This architecture leads to 11 messages in a client cycle. 1 message is used for the sending of the request by the client, 2 messages are sent by the PP-daemon for forwarding the requests to the servers. Each server needs 4 messages (2 related to the method invocation and 2 for client handle verification) for invoking a method in the

client that receives the results of the desired operations, leading to 8 messages in total being exchanged between the client and the servers.

The Synchronous PP-daemon-Based Architecture:

As in case of the forwarding architecture, the request for the entire job is sent to the PP-daemon. The PP-daemon invokes the servers concurrently. The servers reply to the PP-daemon that combines the results and send a reply for the entire job back to the client. This architecture leads to 6 messages in a client cycle: 2 messages are interchanged between the client and the PP-daemon for sending the request and receiving the results and 2 for invoking and receiving the results from each server. Note that handle verification performed in the three other architectures is not required since the PP-daemon is aware of all the server handles.

3. EXPERIMENTAL ENVIRONMENT

Performance prototypes of these 4 architectures are constructed. The E2A ASP version 5.1 CORBA compliant middleware from Iona is used. The prototypes are written in C++ and run on a network of Pentium IV and Pentium II PC's under Linux 7.2. These that are PC's interconnected by a 100 MBPS Ethernet form the "quiet network" in our lab where experiments can run in isolation without any interference from other users. Each client is implemented by a separate thread on a single PC. Since most of the time spent by a client is in waiting for a response to its request, a single PC is adequate for handling the maximum number (10) of clients used in the experiment. Each of the servers and the PP-daemon is allocated on a separate PC. The results of the experiments are expected to be independent of the type of equipment used and the conclusions derived reflect the relative performances of the client-daemon-server interaction architectures that this paper focuses on.

A synthetic workload is used. Such a workload is appropriate for answering "what if" questions that are important in the context of this research. As described in Section 2, each client runs cyclically and performs two method invocations in a cycle for competing a job. The servers burn a predefined amount of CPU time to simulate the service demand. We ran a number of experiments with different workload parameters. A representative set of results is presented in this paper. More data will be available in [4]. The important factors used to control the workload of the experimental system are briefly described.

Number of clients (N): The total number of active clients.

Request Service Demands (DA, DB, DC, DD): Unless mentioned otherwise, a fixed distribution with a given mean (SA, SB, SC, or SD) is used for modelling server execution times. With a fixed distribution the execution time for a server is equal to its mean and does not vary from one invocation to another. For the sake of simplicity we have fixed the server demand SA=SC and SB=SD. To investigate the impact of variability in execution times an exponential distribution is used in the experiments described in Section 4.2

Message Size (L): The length of the message used for sending the request as well as for sending back the reply are represented by L. In each experiment the size of the messages propagating among all participating processes was held at a fixed value.

3.1 Performance Measures

The performance measures of interest are briefly summarized

The Mean Client Response Time (R): This is the mean total elapsed time from the instant the client starts its request cycle to the time the final reply is received from the last server used in the cycle. This parameter is a measure of the latency for the system.

The Overall Mean System Throughput (X): This is the average total system throughput, which is obtained by the summation of the mean throughputs of all the clients. The throughput of a client is the average number of cycles executed per second. This parameter is a measure of capacity for the system. The throughput and response time are related by Little's law: $N = XR$ where N is the mean number of clients in the system.

4. RESULTS OF EXPERIMENTS

A number of experiments for different combinations of the factors is run. Each experiment is run multiple times so as to produce a confidence interval of +- 2.5% at a confidence level of 95% for the desired performance measure of interest. Only a representative set of results is reported in this paper. More results are available in [4].

Figure 1 and Table 1 present the performances achieved with the different architectures. N, the number of clients is the variable factor. DA (DC) and DB (DD) are fixed at 50 ms and 55 ms respectively. The length of the messages L is held at 50 bytes.

Two performance attributes, latency and scalability are of interest. At low load the performance measure of interest is the mean client response time that represents the latency properties of the architectures. At medium to high load the overall mean system throughput is used to study the scalability properties of these architectures.

Table 1. Performance Comparison of the Architectures (DA=DC= 50 ms, DB=DD=55 ms, L =50 bytes)

	S	HP	FP	SP
R (ms): N=1	118.5	64.0	76.6	57.6
X (cycles/sec): N=10	34.6	34.6	27.1	17.2

Low Load:

At N=1, the best performance is achieved with the Synchronous PP-Daemon-based architecture. It demonstrates the shortest response time that produces an over 100% improvement in performance in comparison to the conventional Sequential architecture (see Table 1). All the architectures that deploy parallelism in server execution lead to a shorter response time in comparison to the Sequential architecture. Among the parallel architectures the Synchronous PP-Daemon-based architecture uses the lowest number of messages in a client cycle (see Table 2) and achieves the best performance. The Forwarding PP-Daemon-based architecture uses the highest number of messages

and demonstrates the highest response time at N=1. However, by using concurrent server invocations, the Forwarding PP-Daemon-based architecture demonstrates a superior performance in comparison to the Sequential architecture. Note that although some of these message initiations are concurrent at the application level, they may be serialized at the operating system or network level. Moreover, the total number of messages reflects the message related total resource demand made by an architecture and is thus an important characteristic of the architecture.

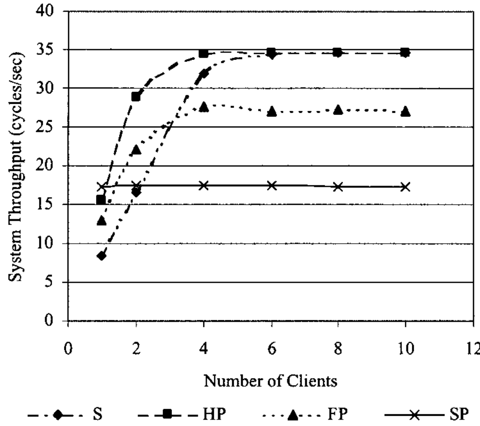

Figure 1. Performance of the Architectures (SA = SC = 50 ms, SB = SD = 55 ms, L = 50 bytes)

Table 2. Number of Messages Used in the Architectures

Interaction Architecture	Number of Messages
Sequential	12
Handle Driven PP-Daemon-based	10
Forwarding PP-Daemon-based	11
Synchronous PP-Daemon-based	6

Medium to High Load:

For most of the architectures, as the number of clients increases the system throughput increases at first and then flattens off at N=6 (see Figure 1). The best performance is demonstrated by the Handle Driven PP-Daemon-based architecture. Although the Forwarding PP-Daemon-based architectures performs the second best initially, it is overtaken by the Sequential Architecture at N=4. The Synchronous PP-Daemon-based architecture that performed the best at N=1 demonstrates a poor performance at high load. At N=10, both the Sequential and The Handle-Driven-PP-Daemon-based architectures attain a throughput of 34.6 cycles/sec whereas the Forwarding PP-Daemon-based architecture attains throughput of 27.1 cycles/sec followed by the Synchronous PP-Daemon-based architecture that achieves a throughput of 17.2 cycles/sec (see Table 1). The rationale for the system behavior is presented next.

The Synchronous PP-Daemon-based architecture processes one job at a time. During the processing of a job the synchronous PP-daemon can remain blocked for a substantial period of time, waiting for the two servers to respond. Once both the replies are available, the PP-daemon sends the result back to the client and then picks up the next job for service. As a result, the PP-daemon process gets saturated at higher loads: it remains busy 100% of the time in processing and waiting for the servers to respond while the underlying CPU is kept idle. This phenomenon is called *software bottlenecking* and a software bottleneck is said to have occurred at the PP-daemon. The software bottleneck introduces large queueing delays to client requests leading to poor system performance. Note that for the parameters used in this experiment the PP-daemon saturates at N>1. For other parameter values, the saturation can occur at higher values of N. The Forwarding PP-Daemon-based architecture performs better than the Sequential architecture at medium load but saturates earlier, at N=4. This is due to the additional work performed by the servers in verifying the client handles in the Forwarding-PP-Daemon-based architecture. An interesting observation is that although the Sequential architecture performs the worst at low load (N=1), its performance improves linearly with an increase in N as it attains the same throughput achieved by the highest performer, the Handle-Driven-PP-Daemon-based architecture.

4.1 Impact of Multi-Threading

An important observation described in the previous section is the formation of performance limiting software bottlenecks. Multi-threading the bottleneck process is an effective way of alleviating the problem. Performances of the three parallel architectures observed with a multi-threaded PP-daemon are presented in Figure 2. Performance of the Sequential architecture is included for the sake of comparison. The Synchronous PP-daemon-based architecture displays a dramatic improvement in performance when the PP-daemon is multithreaded. The performances of the other architectures remain unaltered. This indicates the absence of a software bottleneck at the PP-daemon for these architectures; consequently no further improvement accrues from multi-threading the PP-daemon process. As shown in Figure 2 the Synchronous PP-Daemon-based architecture demonstrates the best performance at all loads. Another important observation made from Figure 1 and Figure 2 is that the benefits of parallel processing are evident at low to medium system load. At high system load the Sequential architecture displays a performance that is comparable to that of the best performing parallel architecture.

4.2 The Effect of Variability in Service Demands

The results of the experiments presented earlier correspond to a fixed distribution for service demands. The impact of variability in service demands as captured by an exponential distribution is presented in Table 3. The mean server demands SA (SC) and SB (SD) are held at 50ms and 55ms respectively. The PP-daemon is multi-threaded. Except for the Forwarding PP-daemon-based architecture, the ranking of the performances of the strategies remains the same as captured in Figure 2: SP performs the best followed by, HP, FP and S. The Forwarding PP-Daemon-based architecture performs much better in comparison to the system with fixed service demands. The benefit that accrues from using

parallelism in server invocation is reflected in the performance difference between SP and S. In comparison to a system with a fixed distribution for service time, the benefit of parallelism as captured in the ratio between X achieved with SP and S for example, seems to be reduced at low load (N=1). At high load (N=10) however, a larger performance difference is observed between SP and S when an exponential distribution-based service demands are used.

Due to space limitations a discussion of the impact of a number of other factors on performance could not be included. A more complete analysis of system performance is presented in [4].

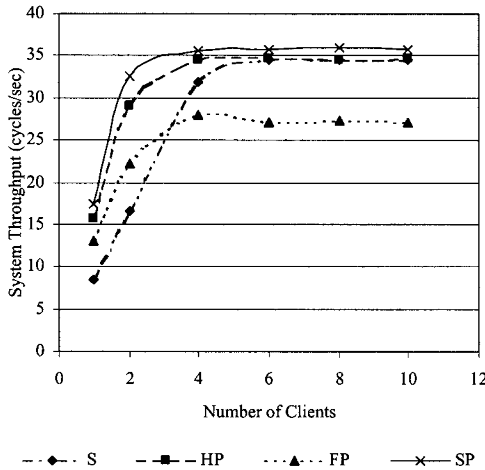

Figure 2. The Effect of Multi-Threading on Performance (SA = SC = 50 ms, SB = SD = 55 ms, L = 50 bytes)

Table 3. System Throughput (cycles/sec) for Exponentially Distributed Service Times (SA=SC= 50 ms, SB=SD=55ms, L=50 bytes)

N	S	HP	FP	SP
1	8.55	11.72	10.56	12.38
4	20.29	24.71	23.79	25.47
10	27.57	30.78	29.17	31.58

5.CONCLUSIONS

Three different parallel interaction architectures are proposed and their performances are compared with that of a traditional sequential architecture. Based on E2A ASP, a COTS CORBA compliant middleware, prototypes are constructed and experiments are run on a network of PC's. The results of a set of experiments have led to valuable insights into system behavior and the relative performances of these architectures. These are briefly summarized.

Performance of the Parallel Architectures: A substantial performance benefit accrues from using a parallel architecture especially at low to medium load. The multi-threaded Synchronous PP-Daemon-based architecture displays the best performance followed by the Handle Driven PP-Daemon-based architecture.

Software Bottleneck: A potential problem with the Synchronous PP-Daemon-based architecture that uses synchronous communication is the formation of a software bottleneck at the PP-daemon. Multi-threading can effectively solve the problem. But on a thread-constrained system, a Handle Driven PP-Daemon-based architecture may be a better choice.

Variability in Service Times: Introducing variability in service times leads to a lower performance benefit for the parallel processing architectures at low load.

This paper presented a preliminary analysis of parallel interaction architectures. A more detailed analysis of the relationship between system performance and workload parameters is presented in [4].

ACKNOWLEDGMENTS

Financial support for this research was provided by the Natural Sciences and Engineering Research Council of Canada, the province of Ontario and Nortel Networks.

REFERENCES

[1] I. Abdul-Fatah, S. Majumdar, "Performance Comparison of Architectures for Client-Server interactions in CORBA," *IEEE Trans. on Parallel and Distributed Systems,* Feb 2002, pp. 111-127.

[2] D. Dhoutat, D. Layimani, "A CORBA-Based Application for Parallel Applications: Experimentation with the WZ Matrix Factorization", Laboratoire de Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France, 2001.

[3] P. Haggerty and K. Seeharman, "The Benefits of CORBA-based Network Management," *Communications of the ACM*, Vol. 41, No. 10. Oct 1998, pp. 73-79.

[4] M. Huo, M.A.Sc. Thesis, Dept. of Systems and Computer Eng., Carleton University, Ottawa, CANADA K1S 5B6 (expected).

[5] IONA Technologies PLC, E2A ASP Users Guide 2002.

[6] Object Management Group, *Data Parallel CORBA Specification*, Nov 2001.

[7] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, ver 3.0.2, Dec 2002.

[8] D. C. Schmidt, D. L. Levine, S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications*, Vol. 21, No. 4, April 1998, pp. 294-324.

[9] J. Siegel, "OMG Overview: CORBA and the OMA in Enterprise Computing," Communications of the ACM, Vol. 41. No. 10, Oct 1998, pp. 37-43.

[10] W. Tao, S. Majumdar, "Application Level Performance Optimizations for CORBA-Based Systems", *Proc. International Workshop on Software and Performance (WOSP02)*, Rome, July 2002, pp. 95-103.

[11] W.-K. Wu, S. Majumdar, "Engineering CORBA-Based Systems for High Performance", *International Conference on Parallel Processing (ICPP02)*, August 2002 Vancouver, pp. 473-482.