PERFORMANCE STUDY OF THE CORBA MULTI-THREADED ORB ARCHITECTURES IN VISIBROKER 3.3

Yuqing Liu

A Project Report

Submitted to

The Graduate Faculty of

Auburn University

In Partial Fulfillment of the

Requirement of the

Degree of

Master in Computer Science and Engineering

Auburn, Alabama

August,1999

ABSTRACT

CORBA (Common Object Request Broker Architecture) provides platform-independent programming interfaces and models for portable distributed object-oriented computing applications. Its independence from programming languages, computing platforms and networking protocols makes it highly suitable for the development of new applications and their integration into existing distributed systems.

CORBA's Object Request Broker (ORB) acts as a software bus for distributed objects. It runs on the top of the operating system and encapsulates the aspects of the underlying operating system and network layers. It delivers client requests to servers and returns responses to the client. To accomplish these tasks, ORBs manage transport connections, perform transport endpoint demultiplexing, invoke distributed objects, and provide the multi-threading architecture used by applications. The architecture used to multi-thread an ORB has a substantial impact on its performance, scalability and predictability. A key challenge for ORB developers and application programmers is to devise threading architectures that can handle multiple client requests efficiently.

Currently, there are six types of the common CORBA multi-threading architectures used by ORB implementations: Thread-per-Request, Thread-per-Connect, Thread-per-Object, Worker Thread Pool, Boss/Worker Thread Pool and Threading Framework. This report will first give a brief overview of multi-threading concepts and the CORBA architecture. Then, the six multi-threaded ORB architectures will be discussed. Finally, the performance of multi-threaded ORB models: Thread-per-Connection and Worker Thread Pool will be evaluated and studied in Visibroker 3.3 in terms of Thread Overhead, Request Traffic Management, Load Balancing and Object Adapter Caching. We conclude that both Thread-per-Connection and Worker Thread Pool multi-threaded ORB architectures have their own advantages and drawbacks. When using properly, they will provide efficiency in achieving performance flexibility, predictability and scalability of client/server applications in a heterogeneous environment.

CONTENTS

1. INTRODUCTION 1
1.1 Benefits of Multi-threaded programming in CORBA 2
1.2 Infrastructure of Multi-threaded ORB 2

2. OVERVIEW OF MULTI-THREADING 3

3. THE COMMON OBJECT REQUEST BROKER ARCHITECTURE 6
3.1 CORBA basics 6
3.2 CORBA ORB Reference model 6

4. THE MULTI-THREADED ORB ARCHITECTURES 10
4.1 The Thread-per-Request Architecture 11
4.2 The Thread-per-Connection Architecture 11
4.3 The Thread-per-Object Architecture 12
4.4 The Worker Thread Pool Architecture 13
4.5 The Boss/Worker Thread Pool Architecture 13
4.6 The Threading Framework Architecture 14
4.7 A Comparison of the Various Multi-threading ORBs 15

5. THE PERFORMANCE STUDY OF MULTI-THREADED ORBs IN VISIBROKER 3.3 17
5.1 Visibroker 3.3 ORB Multi-threading Management 17
5.2 Performance Implementation 20
5.2.1 Latency Measurement 21
5.2.2 Interfaces Definition 21
5.2.3 Client Programs 22
5.2.4 Server Program 24
5.2.5 Database Interface 25
5.2.6 Java Applet Demonstration 27
5.3 Performance Measurements and Results 29
5.3.1 Thread Overhead 29
5.3.2 Request Traffic Management 32
5.3.3 Load Balancing 33
5.3.4 Object Adapter Caching 34
5.4 Conclusion 37

REFERENCES 38

LIST OF FIGURES

Figure 1. Threads in a Process 4

Figure 2. Components in the CORBA ORB Reference Model 8

Figure 3. Thread-per-Request Multi-threading Architecture 11

Figure 4. Thread-per-Connection Multi-threading Architecture 12

Figure 5. Thread-per-Object Multi-threading Architecture 12

Figure 6. Worker Thread Pool Multi-threading Architecture 13

Figure 7. Boss/Worker Thread Pool Multi-threading Architecture 14

Figure 8. Threading Framework Multi-threading Architecture 14

Figure 9. Connection Management 18

Figure 10. Visibroker ORB Request Demultiplexing 19

Figure 11. Client-Server Model 20

Figure 12. TPool vs. TSession server performance with increasing requests 30

Figure 13. TPool vs. TSession server performance with increasing contention 31

Figure 14. TPool vs. TSession server performance in handling request traffic 32

Figure 15. TPool vs. TSession servers performance with load balancing 34

Figure 16. Request Train vs. Round Robin performance in TSession server 35

Figure 17. Request Train vs. Round Robin performance in TPool server 36

1 INTRODUCTION

CORBA (Common Object Request Broker Architecture) provides platform-independent programming interfaces and a model for portable distributed object-oriented computing applications. Its independence from programming languages, computing platforms and networking protocols makes it highly suitable for the development of new applications and their integration into existing distributed systems.

Practical CORBA applications must be able to scale well in several dimensions. These dimensions include the number of objects that an application can support, the number of requests it can handle simultaneously, the number of connections it allows, and the amount of CPU and memory resources it uses.

One important method of making application scale well is to employ multi-threaded programming techniques. Although multiple threads allow true concurrent programming only on multi-processors, using them in simple CPU system can simplify program logic as well as enhance program scalability and performance.

Ordinarily, processes on commonly used operating systems such as Windows NT and Sun Solaris are single-threaded. The single thread of control that runs within the process performs all actions taken by a single-threaded process, from accessing a variable on the run-time stack to sending and receiving network packets through a socket. Unfortunately, it is often difficult to develop and use server applications that use only a single thread of control. As the complexity of the server application increases, it becomes more and more difficult to ensure that all tasks in the system are getting their share of the single thread. Boundaries should be set where one task explicitly yields the single thread to other tasks. Preventing task starvation due to blocking reads becomes more and more difficult, and maintenance becomes complicated because each modification of the program requires analysis to ensure that it does not introduce task starvation.

Single-threaded systems are fine for servers that are used by only a few clients for short duration requests. Pure client applications, which contain no CORBA objects, also work well when single-threaded. High-performance servers, on the other hand, are usually multi-threaded.

1.1 Benefits of Multi-threaded Programming in CORBA

Using multi-threaded programming techniques to implement CORBA server applications provides benefits such as [1]:

- simplifying program design by allowing multiple servants to execute independently using conventional programming abstractions such as synchronous CORBA remote method requests and replies;

- improving end-to-end throughput and latency performance by using the parallel processing capabilities of multi-processors hardware platforms and by overlapping computation with communication;

- improving perceived response time for interactive client applications, such as user interface or network management tools, by associating separate threads with different operations so client operations do not block indefinitely.

1.2 Infrastructure of Multi-threaded ORB

Most portions of a CORBA server application are affected by the use of multi-threading structures, especially, in the multi-threaded ORB implementations. Multi-threaded ORB implementations have a wide variety of available options for handling requests. Single-threaded ORBs must perform non-blocking I/O, request queuing, and explicit task

switching, thus limiting their flexibility and configurability. On the other hand, multi-threaded ORBs can support different strategies for request dispatching simultaneously.

For example, one implementation of a multi-threaded ORB core might have a single thread, called a listener thread that listens for requests. When a request arrives, the thread reads the entire network message containing the request and places it in a request queue. The other end of the queue might have a pool of threads that wait for requests to appear in the queue. When a request is put into the queue by the listener thread, a thread from the pool removes it from the queue and takes charge of dispatching it to the right servant.

Another ORB core implementation might choose to use multiple listener threads, with each thread listening to a single network port. Still another might choose to create a new thread to handle each incoming request. Other variation could mix support for multiple strategies into a single ORB core. Each solution for applying threads to the processing of requests has its own benefits and drawbacks.

Currently, there are six types of the common CORBA multi-threading architectures used by ORB implementations: Thread-per-Request, Thread-per-Connect, Thread-per-Object, Worker Thread Pool, Boss/Worker Thread Pool and Threading Framework. The architecture used to multi-thread an ORB has a substantial impact on its performance, scalability and predictability. A key challenge for ORB developers and application programmers is to devise threading architectures that can handle multiple client requests efficiently. This report will first have a brief overview of multi-threading concept and CORBA architecture. Then, the six multi-threaded ORB architectures will be discussed. Finally, the performance of multi-threaded ORB models: Thread-per-Connection and Worker Thread Pool will be evaluated and studied using Visibroker 3.3.

2 OVERVIEW OF MULTI-THREADING

A thread is a lightweight process that executes a sequence of instructions and reduces overhead by sharing fundamental parts of a program with other threads. Threads are

lightweight so that there can be hundreds of them present within a process. In addition to their own instruction pointers, threads contain resources such as a run-time stack of method activation records, a set of registers, and thread-specific storage. With a preemptive multi-threaded OS, such as Solaris or Window NT,
the underlying operating system kernel or a special threading library controls the scheduling of threads to allow them to execute their tasks. A slice of CPU time is given to each thread. When the thread either uses up its time slice or makes a blocking call such as reading from a socket, the scheduler preempts the threads and allows another one to run. This arrangement relieves program complexity and ensures that all necessary tasks get the CPU time they need to complete. It also allows the application to remain wholly separate, permitting users to implement and maintain them without fear of compromising the correctness of the application because of task starvation.

With threaded programming, multiple tasks can run concurrently within the same program. They can share a single CPU as processes do or take advantage of multiple CPUs when available. This enables applications to be structured efficiently with simultaneous threads servicing independent computations. Likewise, multi-threading can minimize latency and ensure scheduling in real-time systems.

Contemporary operating systems, such as Sun Solaris OS, support the concurrent execution of multiple processes, each containing one or more threads. As shown in Figure 1, a process serves as the unit of protection and resource allocation within a separate hardware-protected address space. A thread serves as the unit of execution running within the process address space that is shared with other threads.

PROCESS

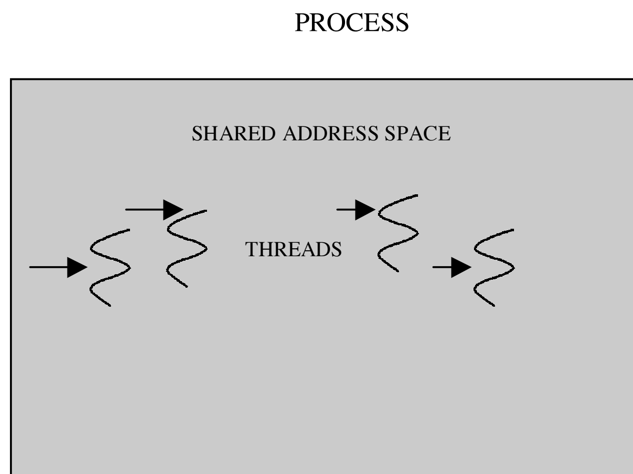


Figure 1. Threads in a Process

The major benefit of multi-threading programs over non-threaded ones is in their ability to execute tasks concurrently. However, in providing concurrency, multi-threading programs introduce a certain amount of overhead. Some of the threading costs are:

1) Cost of the shared data resources: Shared data and associated locks are both the greatest asset and the biggest curse in multi-threading programming [2]. That threads in the same process have equal access to a common set of resources, including the process's address space, allows them to communicate with each other much faster than independent processes can. However, as more thread resources are shared, the more the performance is diminished. Locks reveal the dependencies among the threads in multi-threaded programs. The impact of each lock on a program's performance is twofold: the time it takes for a thread to obtain an un-owned lock; the time a thread spends while waiting for a lock that is already held by another thread.

2) Thread overhead: Although the cost of creating and synchronization multiple threads is less than that of spawning and coordinating multiple processes, using threads nevertheless does involve overhead. When creating a thread, the thread library (and perhaps the system) must perform database searches and allocate new data structures, synchronizing the creation of this thread with other threads that may be in progress at the same time. The runtime system must place the newly created thread into the system's scheduling queues. In a kernel thread-based implementation, this requires a system call. The result is that operating system allocates resources for the threads that are similar to those it allocates for a process.

3) Thread context switches: Once threads have been created, they must share the limited CPU resources. Regardless of whether a user space or kernel thread-based implementation is used, scheduling a new thread requires a context switch between threads. The running thread is interrupted and its registers and other private resources are saved. A new thread is select from the scheduler's priority queues, and its registers and private context are brought in from swap space. As more threads are added to a program, the context switches will cost more in performance.

4) Synchronization overhead: Each synchronization object (a mutex, condition variable, etc.) requires that the thread library creates and maintains some data structures and execute some code. Consequently, creating large numbers of synchronization objects has its own cost. This cost can be magnified by the way in which the synchronization objects are deployed. For instance, if one creates a lock for each record in a database, the disk space required to store the database will be increased as well as the memory required holding the lock while a thread is running.

3 THE COMMON OBJECT REQUEST BROKER ARCHITECTURE

3.1 CORBA Basics

The Common Object Request Broker Architecture (CORBA) is a communication facility designed specifically for distributed objects. It is a layer of abstraction that hides differences in heterogeneous environments. The CORBA standards were first published in 1991 by the Object Management Group (OMG) [3]. The core of the CORBA architecture is the Object Request Broker (ORB) that acts as an object bus over which objects transparently interact with other objects located locally or remotely [4]. The services that a CORBA object can provide are expressed in an interface with a set of methods. A particular instance of object is identified by an object reference. The client of a CORBA object acquires an object reference and uses it as a handle to make method calls [5].

Two aspects of the CORBA architecture stand out:

• Both client and object implementations are isolated from the Object Request Broker by an IDL interface. CORBA requires that every object's interface be expressed in the Object Management Group's Interface Definition Language (OMG IDL). Clients see only the object's interface, but not the implementation details. This guarantees substitutability of the implementation behind the interface.

• A service request does not pass directly from client to object implementation. Instead, requests are always managed by ORBs. Every invocation of a CORBA object is passed to the ORB; the form of the invocation is the same whether the target object is local or remote. Distribution details remain in ORBs where they are handled by the software.

3.2 CORBA ORB Reference Model

The Object Request Broker defines the CORBA object bus. It resides in the center of the CORBA distributed object system. The ORB is responsible for all the mechanism required to publicize an object, find an object's implementation, prepare it to receive the request, communicate the request to it, and carry the reply back to the requesting client. All these features allow clients to invoke operations on distributed objects without concern for the following issues:

Object location transparency: CORBA objects either can be co-located with the client or distributed on a remote server, without affecting their implementation or use.

Programming language interoperability: The languages supported by CORBA include C, C++, Java, Ada95, COBOL, and Smalltalk, among others. The programming language used in client applications and server implementations can be different.

Operating system platform portability: CORBA runs on many OS platforms, including Window NT, UNIX, MVS, and real-time embedded systems like VxWorks, Chorus, and LynxOS.

Communication protocols and interconnects: The communication protocols and interconnects that CORBA can run on include TCP/IP, IPX/SPX, FDDI, ATM, Ethernet, Fast Ethernet, embedded system backplanes, and shared memory.

Hardware: CORBA shields applications from side effects stemming from differences in hardware, such as storage layout and data type sizes/ranges.

Figure 2 illustrates the components in the CORBA 2.x Reference Model. All those components collaborate to provide the portability, interoperability, and transparency described above. The components in the CORBA reference model shown in Figure 2 are described below:

• Objects: In CORBA, an object is an instance of an Interface Definition Language (IDL) interface. The object is identified by an *object reference*, which uniquely names that instance across servers. An *ObjectID* associates an object with its servant implementation, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

• Client: A client is a program entity that performs application tasks by obtaining object references to objects and invoking operations on them. Objects can be remote or co-located relative to the client. Ideally, accessing a remote object should be as simple as calling an operation on a local object.

• Server: A server is an application in which one or more CORBA objects exit. As with clients, this term is meaningful only in the context of a particular request.

• Servant: The servant implements the operations defined by an OMG Interface Definition

Language (IDL) interface. In procedural language, servants are typically implemented using functions and structures. In object-oriented programming (OOP), servants are

implemented using one or more class instances. A client never interacts with a servant directly, but always through an object identified by an object reference.

• ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for
delivering the request to the object and returning a response, if any, to the client. For an object

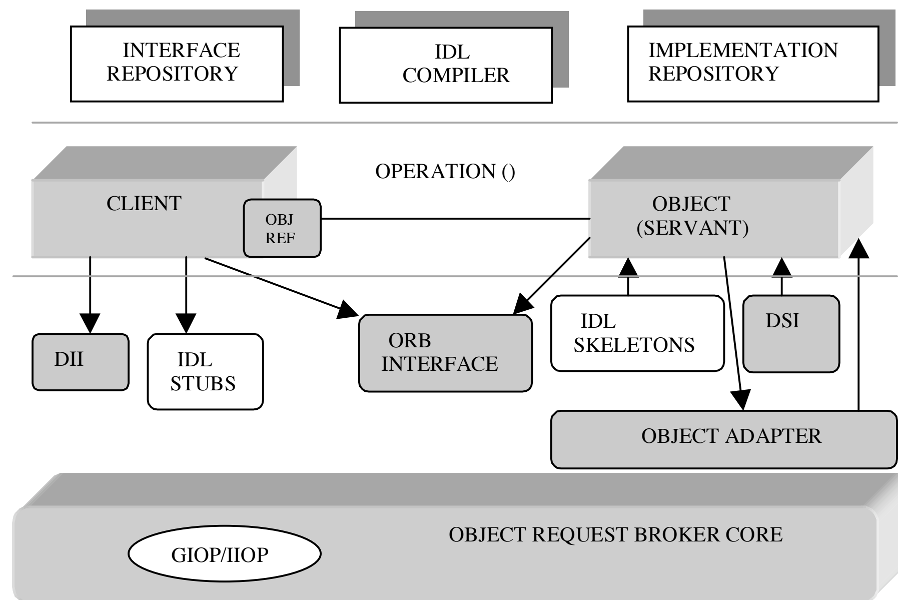


Figure 2. Components in the CORBA ORB Reference Model

executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), most commonly the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. An ORB Core is typically implemented as a run-time library linked to client and server applications.

• ORB Interface: An ORB is an abstraction that can be implemented various ways, e.g., one or more processes or a set of libraries. To de-couple applications from implementation details, the CORBA specification defines an interface to an ORB. This ORB interface provides standard operations to initialize and shutdown the ORB, converts

object references to strings and back, and creates argument lists for requests made through the dynamic invocation interface (DII).

• IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and server, and the ORB. Stubs provide a strongly typed, static invocation interface (SII) that marshals application parameters into a common data-level representation. Conversely, skeletons de-marshal the data-level representation into typed parameters that are meaningful to an application.

• IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimization [6].

• Dynamic Invocation Interface (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 in accesses. The DII also allows clients to make deferred synchronous calls, which de-couple the request and response portions of two-way operations to avoid blocking the client until the server responds.

• Dynamic Skeleton Interface (DSI): The DSI is the server's analogue to the client's DII. The DSI allows an ORB to deliver requests to a server that has no compile-time knowledge of the IDL interface that the server implements. Clients making requests need not know whether the server ORB uses static skeletons or dynamic skeletons. Likewise, servers need not know if clients use the DII or SII to invoke requests.

• Object Adapter: An Object Adapter associates servers with objects, creates object references, demultiplexes incoming requests to servers, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a server. Object Adapters enable ORBs to support various types of servers that possess similar requirements. This design

results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles and other properties.

• Interface Repository: The Interface Repository provides run-time information about IDL interfaces. Using this information, it is possible for a program to discover an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make invocation on it using the DII. In addition, the Interface Repository provides a common location to store additional information associated with interfaces to CORBA objects, such as type libraries for stubs and skeletons.

• Implementation Repository: The implementation repository contains information that allows an ORB to activate servers to process servants [7]. Most of the information in the IR is specific to an ORB or OS environment. In addition, the IR provides a common location to store information associated with servers, such as administrative control, resources allocation, security and activation modes.

4 THE MULTI-THREADED ORB ARCHITECTURES

Programming applications with CORBA ORBs without multi-threading is hard, particularly to develop servers or real-time applications. Without multi-threading capabilities, developers must ensure that requests can be handled quickly without starvation. In practice, many requests cannot be serviced quickly enough in a single-threaded ORB to avoid starving clients.

With multiple threads, each request can be serviced in its own thread, independent of other requests. This way, clients are not starved when waiting for their requests to be serviced. Likewise, system resources are conserved, since creating a new thread is typically much less expensive than creating an entirely new process.

There are a variety of strategies for structuring a multi-threading architecture in an ORB. A number of alternative ORB Core multi-threading architectures are described below, focusing on server-side multi-threading.

4.1 The Thread-per-Request Architecture

The HP ORBPlus ORB uses the Thread-per-Request architecture. In this architecture, a separate thread of control handles each request from a client. The components of the Thread-per-Request architecture include an I/O thread and one or more dynamically spawned threads. The I/O thread simply manages the receipt of incoming requests. After it obtained a request from the socket endpoint, the I/O thread will spawn a thread to which it will pass the request. The spawned thread will handle request-processing upcall to the server. When the upcall complete, the spawned thread exits. Figure 3 shows the Thread-per-Request architecture.

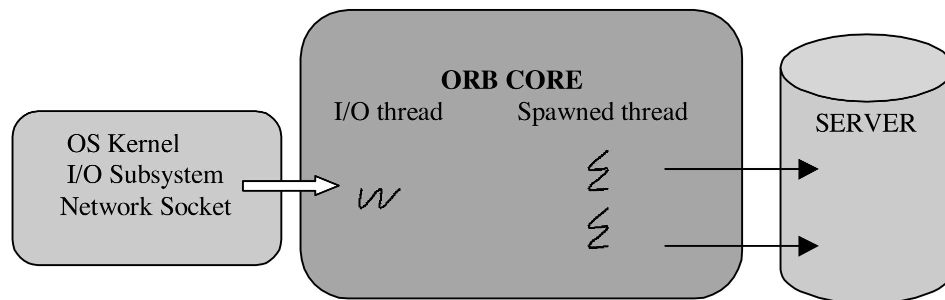


Figure 3. Thread-per-Request Multi-threading Architecture

4.2 The Thread-per-Connection Architecture

Inprise Visibroker 2.0 and SunSoft IIOP implement the Thread-per-Connection architecture.

The Thread-per-Connection architecture is a variation of Thread-per-Request that amortizes the cost of spawning the thread across multiple requests from the same client

process. As shown in Figure 4, the components in the Thread-per-Connection architecture are a set of connection threads, each of which is dedicated to handle a separate client for the duration of its connection. Each connection thread obtains a new request directly from its socket endpoint, dispatches the request to the server, and then returns to read the next request from its connection. When the connection is broken, the thread is discarded.

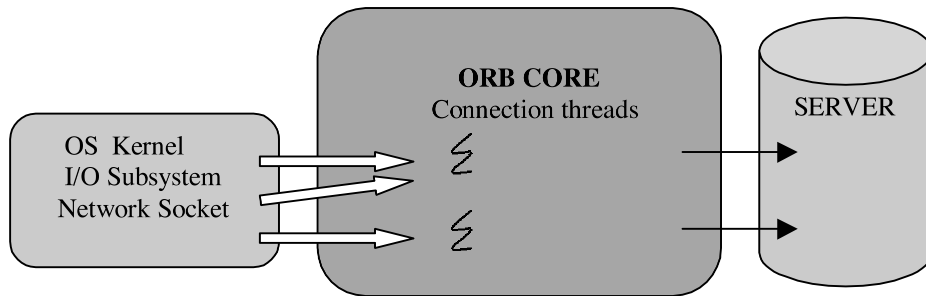


Figure 4. Thread-per-Connection Multi-threading Architecture

4.3 The Thread-per-Object Architecture

MT-Orbix can be configured to support Thread-per-Object. The Thread-per-Object architecture

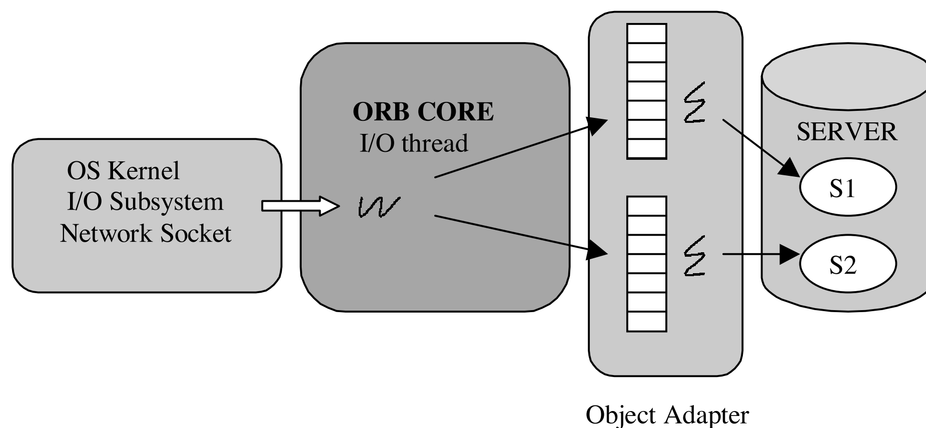


Figure 5. Thread-per-Object Multi-threading Architecture

associates a thread for each object [8]. As shown in the Figure 5, the components in the Thread-per-Object architecture are an I/O thread and a set of threads. Each thread is dedicated to handle a separate object's servant, e.g., S1, S2, and S3. The I/O thread simply manages the incoming requests. It reads a new request from a socket endpoint and passes it to the Object Adapter. The
Object Adapter then inserts the request into a queue associated with a servant and the servant's thread. This servant's thread will dequeue request from its queue, and dispatch the upcall on the servant.

4.4 The Worker Thread Pool Architecture

Visibroker 3.3 ORB from Inprise implements the Worker Thread Pool architecture. A thread pool is another variation of the Thread-per-Request architecture that amortizes thread creation costs by pre-spawning a pool of threads. As shown in the Figure 6, the components in a Worker Thread Pool include an I/O thread, a request queue and a pool of worker threads. The I/O thread selects on

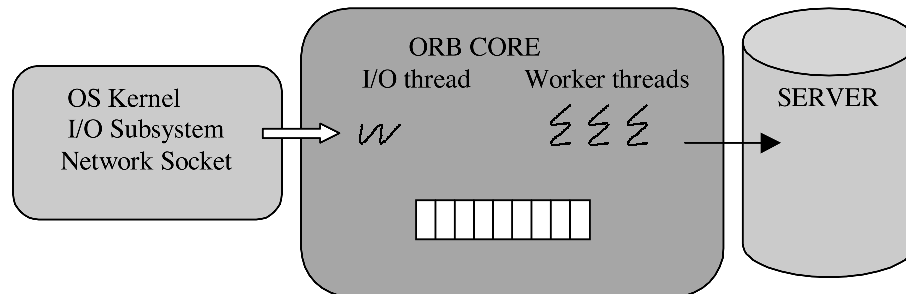


Figure 6. Worker Thread Pool Multi-threading Architecture

the socket endpoints, reads new client requests, and inserts them into the tail of the request queue. A worker thread in the pool dequeues the next request from the head of the queue and dispatches it to the server.

4.5 The Boss/Worker Thread Pool Architecture

Sun miniCOOL ORB uses the Boss/Worker Thread Pool architecture. The Boss/Worker Thread Pool architecture is an optimization of the Worker Thread Pool model. As shown in Figure 7, a pool of threads is allocated and a boss thread is chosen to select on connections for all servants in the server process. When a request arrives, this thread reads the request into an internal buffer. If this is a valid request for a servant, a worker thread in the pool is released to become the new boss and the boss thread dispatches the upcall. After the upcall is dispatched to the server, the original boss thread becomes a worker thread and returns to the thread pool. New requests are queued in socket endpoints until a thread in the pool is available to execute the requests.

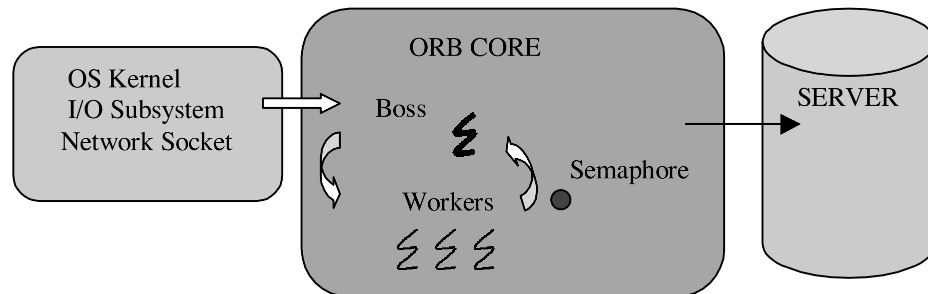
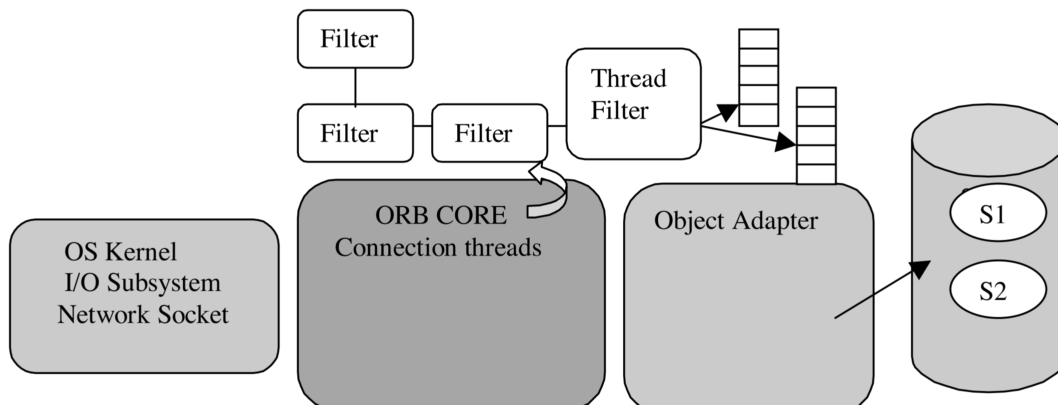


Figure 7. Boss/Worker Thread Pool Multi-threading Architecture

4.6 The Threading Framework Architecture

In the MT-Orbix multi-threading ORB implementation [9], a very flexible architecture called Threading Framework is used. An application can install a thread filter at the top of chain filters. Filters are application-programmable hooks that can perform a number of tasks. Common tasks



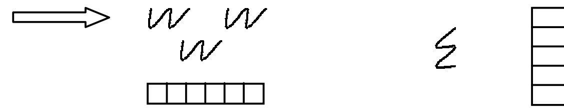


Figure 8. Threading Framework Multi-threading Architecture

include intercepting, modifying, or examining each request sent to or from the ORB. Figure 8 shows one way of structuring this framework. In the Threading Framework architecture, a
connection thread reads a request from the socket endpoint and enqueues the request on a request queue in the ORB core. Another thread then dequeues the request from the request queue and successively passes it through each filter in the chain. The topmost filter, a thread filter, determines the thread to handle this request. The thread filter can be used to implement various multi-threading strategies. For example, using the Thread Pool model, the thread filter enqueues the request into a queue serviced by a thread with the appropriate priority. This thread then passes control back to the ORB, which performs operation demultiplexing and dispatches the upcall to the server.

4.7 A Comparison of the Various Multi-threaded ORBs

Generally, these six ORB multi-threading architectures are commonly used by one or more CORBA implementation. Each has its advantages and disadvantages.

As we have seen, it is straightforward to implement the Thread-per-Request architecture. But with Thread-per-Request, an implementation can consume a large number of OS resources if many clients make requests simultaneously. Moreover, it is inefficient for short-duration requests because it incurs excessive thread creation overhead. In addition, Thread-per-Request architectures are not suitable for real-time applications because the overhead of spawning threads for each request is non-deterministic.

The Thread-per-Connection architecture is a variation of Thread-per-Request that amortizes the cost of spawning the thread across multiple requests from the same client process. As comparing to Thread-per-Request, a Thread-per-Connection has a simple

implementation. It is well suited for an ORB that performs long-duration conversations with multiple clients. The primary disadvantage with Thread-per-Connection is that it does not handle incoming requests more effectively. Moreover, for clients that make only a single request to each server, a Thread-per-Connection is equivalent to the Thread-per-Request architecture.

The Thread-per-Object architecture associates a thread for each object. A Thread-per-Object is useful for programmers who want to minimize the amount of rework required to provide multi-threading to existing single-threaded servants. As long as all methods in a servant only access servant-specific state there is no need for explicit synchronization operations. The reason the Thread-per-Object does not handle incoming requests form clients effectively is that it serializes request processing in each servant. Therefore, if one servant receives considerably more requests than another does, there is a performance bottleneck.

The Thread Pool is another variation of the Thread-per-Request architecture that amortizes thread creation costs by pre-spawning a pool of threads. The Thread Pool architecture is useful for an ORB that wants to bind the number of OS resources it consumes. Client requests can be executed concurrently until the number of simultaneous requests exceeds the number of threads in the pool. At this point, additional requests must be queued until a thread becomes available.

A Thread Pool is a common architecture for structuring ORB multi-threading, particularly for real-time ORBs. There are two kinds of thread pools: Worker Thread Pools and Boss/Worker Thread Pools. The main advantage of the Worker Thread Pool multi-threading architecture is its simplicity of implementation. In particular, the request queue provides a straightforward producer/consumer design. The disadvantages of this model stem from the excessive context switching and synchronization required to manage the request queue. Compared with the Worker Thread Pool design, the main advantage of the Boss/Worker Thread Pool architecture is that it minimizes context-switching overhead incurred by incoming requests. Overhead is minimized because the

request does not need to be transferred from the thread that read it to another thread in the pool that processes it. The Boss/Worker architecture's disadvantages are largely the same as with the Worker Thread Pool design. In addition, it is harder to implement the Boss/Worker model than the Worker Thread Pool model.

The Threading framework has more flexibility because the thread filter mechanisms can be incorporated into server process to support various multi-threading strategies. For instance, to implement a Thread-per-Request strategy, the thread filter can spawn a new thread and pass the request to it. Likewise, the MT-Orbix Threading Framework can be configured to implement other multi-threading architectures such as Thread-per-Object or Thread Pool. In addition, thread filter can be used to achieve the load balancing by dispatching the upcall to the servers evenly. However, with a Threading Framework, its generality can significantly increase locking overhead. For instance, locks must be acquired to insert requests into the queue of the appropriate thread in the thread pool. The overhead from locking can greatly reduce throughput and increase latency.

5 THE PERFORMANCE STUDY OF MULTI-THREADED ORBS IN VISIBROKER 3.3

Visibroker 3.3 provides a complete CORBA 2.0 ORB runtime and supporting development environment for building, deploying and managing distributed applications that open, flexible, and interoperable across platforms.

5.1 Visibroker 3.3 ORB Multi-threading Management

Visibroker 3.3 provides two thread models for multi-threaded server [10]:

- Thread-per-Session
- Thread Pool

1) Thread per Session uses the Thread-per-Connection architecture. A new worker thread will be

allocated each time a new client connects to the server. A worker thread will be assigned to handle all the requests received from this client's connection. When the client disconnected from the server, the worker thread exits.

2) Thread Pool uses the Worker Thread Pool architecture. A thread is assigned for each client request, but only for the duration of that particular request. When a request is completed, the worker thread that was assigned to that request is placed into a pool of available threads so it can be reassigned to future client requests.

Each of these models is implemented as a specific Basic Object Adapter (BOA). In an application, the selection of a specific threading model is simple. The thread modeling policy can be explicitly defined as signatures to the BOA_init(argc, argv) function call ("TPool" or "TSession"). The default threading policy is TPool. The threading model can also be set on a command-line when launching the server application using the <–OAid> option. The maximum number of threads in a pool can be set using the BOA::thread-max() call. This call also has a command-line equivalent by using the <-OAthreadMax> option.

Visibroker provides connection management between clients and servers in the ORB core, based on the thread policies selected by each. Overall, Visibroker's connection management ensures all client requests are multiplexed over the same connection, even if they originate from different threads. As Figure 9 shows, two client processes make requests to the server. The top client process is bound to two different objects A and B in the server process. Each of its bind() call requests share a common connection to the server process. The bottom client process is a

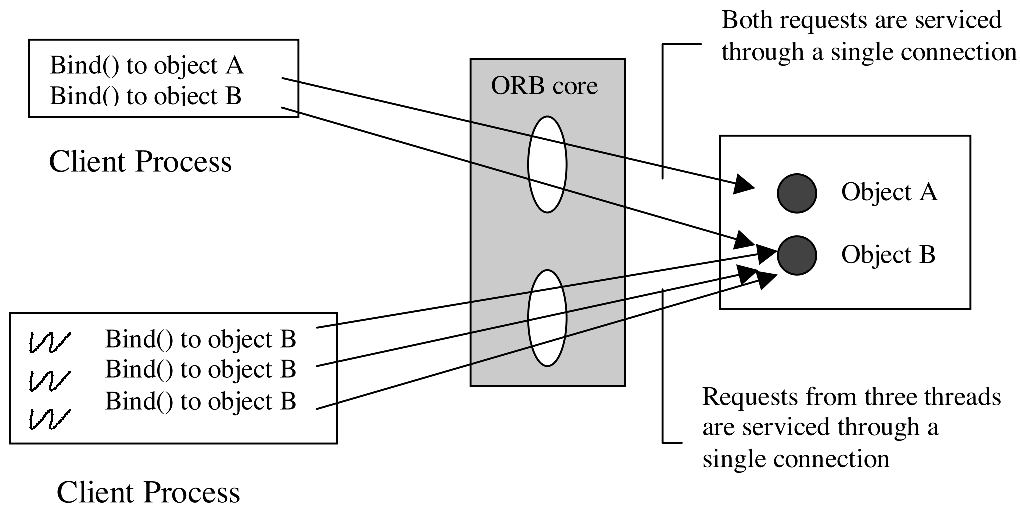


Figure 9. Connection Management

multi-threaded client and it has single connection with several threads bound to the same object B in the server process. All binds from all threads are serviced by the same connection.

On the server side, Visibroker ORBs demultiplex client requests to the appropriate operation of the target object implementation [11]. The progress of Visibroker ORBs demultiplex client requests to the appropriate operation of the servant is shown in Figure 10. The OS protocol stack demultiplexes the incoming client request through the network adapter (data link, network, transport layers) and OS I/O subsystem to the ORB's Object Adapter. The Object Adapter uses the address information in the client request to locate the appropriate target object implementation and associated IDL skeleton. The skeleton locates the appropriate operation, demarshals the request buffer into the operation parameters, and performs the upcall to the operation.

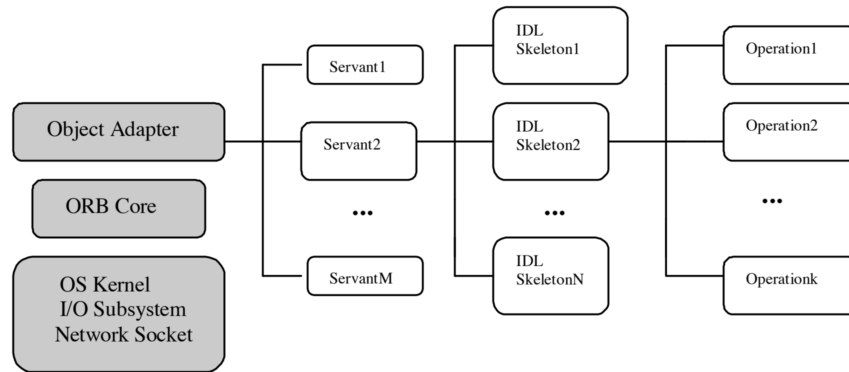


Figure 10. Visibroker ORB Request Demultiplexing

When clients simultaneously talk to servers, Visibroker provides a Smart Agent to locate the object implementations that the client programs wish to use. The Visibroker Smart Agent (osagent) is a dynamic, distributed directory service that provides facilities used by both client programs and object implementations. A Smart Agent must be started on at least one host within the local network. On the client side, when the bind() method on an object is invoked by the client program, the Smart Agent is automatically consulted. On the server side, when a globally scoped object implementation invokes either the BOA::obj_is_ready() method or the BOA::impl_is_ready() method, the Smart Agent registers the object or implementation so that it can be used by client programs. When a client send a request to a server, the Smart Agent locates the specified implementation in order to have a connection established between the client and the implementation. The communication with the Smart Agent is completely transparent to the client program.

Visibroker locates a Smart Agent for use by a client program and object implementation using a broadcast message. Smart Agent can be started on more than one host in the local network. The first Smart Agent to respond is used. After a Smart Agent has been located, Visibroker uses a point-to-point UDP connection to send registration and look-up requests to the Smart Agent.

To start a Smart Agent on a host in the local network, no special coding techniques are required. The basic command for starting the Smart Agent is as follows:

```
prompt> osagent &
```

When more than one server is connected to the Smart Agent, all incoming requests from clients will distributed to the servers based on the current server's workload to achieve load balancing.

5.2 Performance Implementation

The following Figure11 is the structure of the client-server model implementation used to do performance study:

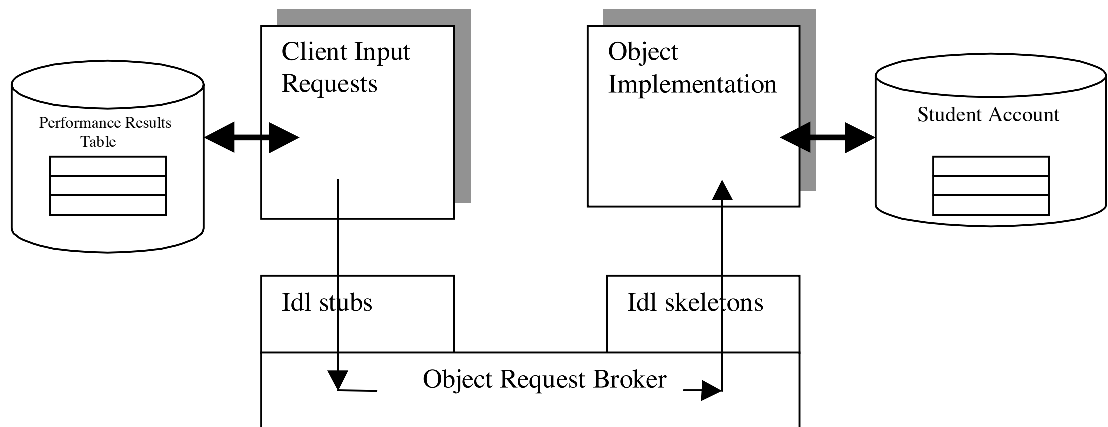


Figure 11. Client-Server Model

There are three parts of implementation logic on the client side: sending request to the ORB server and waiting for a respond from the server; performance measurement-related calculation; and storing the results of calculation into a database table. There are also three main parts of logic on the server side: waiting for a request from client; object implementers for objects invoked by the clients; connecting to the database and storing data for contention case study.

In our implementation, the client and server programs run under the SUN Solaris2.5 Operating System platform. The software used to build both client and server programs is CORBA Visibroker 3.3 for C++, Oracle 7.2.2, Pro* C/C++ 2.1.2, SUN OS SC4.0 C++, Java 1.0 and the JDBC library draft version.

5.2.1 Latency Measurement

Latency is used to measure the performance of the two multi-threaded model servers that were described in section 5.1. Latency is the amount of time necessary for round trip message travel between client and server. There are three main factors that affect the latency: object transit time, application processing time and communication delay. Object transit time is the amount of time it takes for a client Sending Request to a client Receiving Request. Application processing time is any processing time that is attributable to the application. Communication time is any delay along the client and server communication path. The time involved for a cross-machine object invocation introduces any delays imposed by the communication path. In this performance study, the client-server communication between two processes is on the same machine so the unpredictable network communication overhead is eliminated.

5.2.2 Interfaces Definition

The object interfaces implemented by the server are defined in the Interface Definition Language. There are four interfaces used in the performance study, which are shown in the followings:

```
interface DataTransaction {
        short ClientSendToServer(in Buffer data);
        short QueryDatabase (in long id, inout Record record); };

interface DataSend1 {
        short ClientSendToServer1(in long data); };

interface DataSend2 {
        short ClientSendToServer2(in long data); };

interface DataSend3 {
        short ClientSendToServer3(in long data); };
```

The *DataTransaction* interface provides two methods: ClientSendToServer() and QueryDatabase(). The ClientSendToServer() method allows a client send a stream of character buffer to the server. The QueryDatabase() method allows a client to obtain a student record from the database. Three interfaces: DataSend1(), DataSend2(), DataSend3() allow a client to send an integer to the server.

In addition to the interfaces, the IDL file also defines a structure used as user-defined data type.

5.2.3 Client Programs

The client performs these steps:

- Initializes the ORB
- Binds to the object
- Invokes the methods on the remote objects, using the object reference returned by the _bind() method
- Gets the results and calculate the latency

The first task of the client program is to initialize the ORB object runtime.

```
int main (int argc, char* const* argv) {
    try {
        // initializing the ORB
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        …
    } catch (…) {}
}
```

Before the client program can invoke the operations declared in the IDL file, it must first invoke the _bind() method. The _bind() method asks the ORB to locate object and establish a connection to the server.

The performance measurements conducted for this project use three different request invocation algorithms on the client side. Each invocation algorithm will be used in evaluating two multi-threaded models on the server side. These three algorithms are Request Chain, Request Train, and Round Robin.

1) Request Chain Algorithm: This algorithm allows the client sends a series of requests to the same servant.

```
for (CORBA::Ulong i=0; i< number_request; i++) {
// call the same remote servant
.... // servant
}
```

The number_request value can be either hard coded or input from command line. The client consequently accesses the same servant until the number_request completes.

2) Request Train Algorithm: This algorithm allows the client sends a series of requests to different
servants in a sequential fashion.

```
for (CORBA::Ulong i=0; i< number_request; i++) {
 .... // servant1
}
for (CORBA::Ulong i=0; i< number_request; i++) {
.... // servant2
}
for (CORBA::Ulong i=0; i< number_request; i++) {
.... // servant3
}
```

2) Round Robin Algorithm: This algorithm allows the client sends a series of requests to different
servants with different object invocations.

```
for (CORBA::Ulong i=0; i< number_request; i++) {
.... // servant1
```

```
.... // servant2
.... // servant3
}
```

The latency is measured in the client programs.

```
struct timeval tstarttime, tendtime;
VISPortable::vgettimeofday(&tstarttime);

// client sends requests to the server and gets response from the server
....
VISPortable::vgettimeofday(&tendtime);
double msecs = (1000.0 * (tendtime.tv_sec - tstarttime.tv_sec)) +
               ( (tendtime.tv_usec - tstarttime.tv_usec) / 1000.0); // time in
milliseconds
```

The UNIX data structure *timeval* is used to measure the latency. The function *vgettimeofday()* is a Visibroker API to get the current time.

5.2.4 Server Program

The server program performs these steps:

- provides the object implementations,
- initializes the ORB runtime,
- instantiates the object and registers the object with the ORB,
- enters a loop waiting for the client requests

Before instantiating an object, the server main routine makes two calls— one to the ORB and the other to the Basic Object Adaptor (BOA). The BOA is the interface between the object implementation and the ORB. The BOA allows objects to notify the ORB when it is ready to accept client requests and informs it when client requests are received:

```
int main (int argc, char* const* argv) {
    try { // Initializing the ORB and BOA
```

```
                CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
                CORBA::BOA_var boa = orb->BOA_init(argc, argv );
                …
          }   catch(…) {}
    }
```

The selection of threading model comes from the command-line input when launching the server application with <– OAid> option:

```
prompt> server –OAid TPool or
prompt> server –OAid TSession
```

5.2.5 Database Interface

Both client program and server program can connect to the Oracle Database. The database API is a Pro*C/C++ interface and used in the Visibroker client/server programs. The insert and query data functions are invoked from client program. Particularly, the query function is invoked through the object invocation to the server and the result will return from the server through database API call. These APIs contain four functions: open_database(), close_database(), insert_data() and query_data(). The following excerpt coding defines the connection to Oracle database in the open_database() function:

```
EXEC SQL BEGIN DECLARE SECTION;
        char *uid = "ops$userid/password";   /* define login host variable */
EXEC SQL END DECLARE SECTION;
/* Connect to ORACLE.   */
EXEC SQL CONNECT :uid;
```

The following excerpt coding defines the disconnection from Oracle database in the close_database() function:

```
/* Disconnect from ORACLE. */
EXEC SQL COMMIT WORK RELEASE;
```

The following excerpt coding shows how to insert data into a Oracle database table in the insert_data() function:

```
EXEC SQL BEGIN DECLARE SECTION;
        char sqlstmt[MAX_BUFFER_SIZE];  /*define host variable */
EXEC SQL END DECLARE SECTION;

sprintf(sqlstmt, "INSERT INTO %s VALUES ('%s',%d, %4.2f)",
        record.TableName, record.ServerType, record.BufferSize,
record.TotalTime);

/* execute the sql statement */
EXEC SQL EXECUTE IMMEDIATE :sqlstmt;
```

The following excerpt coding shows how to retrieve the data from Oracle database in the query_data() function:

```
EXEC SQL BEGIN DECLARE SECTION;
  int  student_id;
  struct data_record {      /* define host structure */
        long StudentId;
        varchar StudentName[UNAME_LEN];
        float gpa;
   } data_record;
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT *   /* retrieve data into a host structure */
INTO :data_record
FROM student_record
WHERE STUDENTID = :student_id;
```

The Oracle 7.2.2, Pro*C/C 2.2.1 does not have thread safe features [12]. As consequences, in our performance study for the contention scenarios, the shared resources must be protected by Mutex
and its locking methods in the server program.

```
//define global mutex variable
VISMutex db_mtx;
…
db_mtx.lock();
int retval = query_data( id, s_record );  // shared resources

db_mtx.unlock();
```

VISMutex is the Visibroker class.

Two tables are created for the performance study. On the client side, the timer_record table is used to store the performance measurement results. On the server side, student_record table is used to query the student record. There are three attributes in the *timer_record*: ThreadType, BufferSize and TotalTime. There are three attributes in the *student_record*: StudentID, StudentName and GPA.

5.2.6 Java Applet Demonstration

Part of our performance study is demonstrated in the Java Applet graphic presentation. The purpose of this Applet is to provide a visualization of our performance study of the multi-threaded CORBA application. The Java program consists of five main parts: applet initialization, event handling, database connection, data retrieve and data paint.

Applet Initialization

The program has several levels in its Components hierarchy. At the top, the Applet has the Frame created in its main() method. Under the Frame is the Client object, which is inherited from Applet. Directly under the Client Object are Canvas and Panel. The Canvas is a subclass to display any kind of drawing. The Panel is to hold an interface within a parent frame to create a button component and to capture the user event actions.

The Layout Manager, BorderLayout(), is used to arrange components along the edges and in the center of its container.

The first part of Applet initialization is to load the JDBC library in advance before accessing JDBC API at run time.

```
try {
    System.loadLibrary("pbj"); // pbj is the JDBC library name
} catch (UnsatisfiedLinkError e) { …  }
```

Event Handling

A Redraw Curve button establishes an interaction between user mouse click and the Applet. Each time the users click the button, the load balancing performance data will be redrawn based on the data stored in the timer_record database table.

```
public boolean action( Event ev, Object arg )
{
        if ( ev.target instanceof Button ) {
                String label = (String) arg;
                if ( label.equals("REDRAWING CURVE") ) {
                        if ( parent.queryDataFromTable() )
                            canvas.redraw( parent.v );
                }
                return true;
        }
         …
}
```

Database Connection

In the Java1.0 JDBC package, the method getConnection() is used to connect to the Oracle database. It takes database name, user id and password as input parameters.

```
try {
```

```
        connection = Environment.getConnection(
                "T:oracle7:csedb", "ops$userid", "password" );
    } …
```

Database Retrieve

To retrieve data from Oracle database, we construct the SQL select statements via the JDBC APIs. Statement.executeQuery(), in which it returns results as rows of data in a ResultSet Object. The results are examined row-by-row using the ResultSet.next() and extracted by ResultSet.getXXX().

```
    Statement statement = connection.createStatement();
    String selectstmt = "select * from TIMER_RECORD";
```

To execute the SQL query:

```
    ResultSet results;
    results = statement.executeQuery( selectstmt );
```

To extract data row-by-row from results array in the order of timer_record table attributes:

```
    while ( results.next() )
    {
        Record record = new Record();
        record.serverType =  results.getChar(1);
        record.numberOfBytes = results.getInteger(2);
        record.totalTime = results.getChar(3);
    }
```

Data Paint

The data is presented in a X-Y coordinate system. The applet does all of its drawing in the paint() method. There are three API methods used for curve drawing: drawLine(), drawRect() and drawString(). drawLine() draws a line based on the value of input X and Y. drawRect() draws a rectangle based on the value of X, Y, width and height. drawString() draws a string at a given X and Y coordinate.

5.3 Performance Measurements and Results

Various multi-threading architectures have been implemented in the CORBA multi-threading ORBs. The purpose is to build more efficient, flexible and predictable client/server applications. In this section, performance of the TPool and TSession multi-threading models will be studied in terms of Thread Overhead, Request Traffic Management, Load Balancing and Object Adapter Caching.

5.3.1 Thread Overhead

Context-switching overhead and synchronization overhead are the major threading costs when building multi-threading programs. In theory, the Worker Thread Pool model consumes more context switching than the Thread-per-Connection does. When a single thread handles all incoming requests from a client in a Thread-per-Connection model, one or more threads can serve for a single client in a Worker Thread Pool model. Scheduling a new thread requires a context switch among threads. When some requests are from different concurrent clients to the shared data resources, there is a contention among threads. When all requests from all clients are aimed at the same object, the threads lose all concurrency.

1) Performance depends on the duration of the client and server conversation.

We tested the capabilities of two multi-threaded ORB servers by increasing the duration of a client and server conversation. We ran 10 concurrent clients in the same machine. Each

client used the Request Chain Algorithm to send its requests to the ORB server. The number of different requests for each client ranged from 10 to 100, in increments of 10. Each concurrent client sent a buffer of 1000 characters to the server for object invocation.

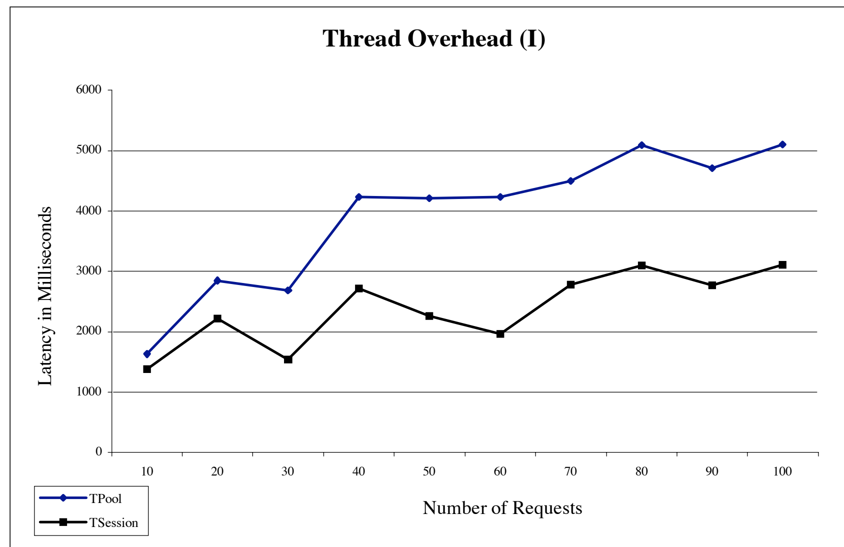


Figure 12. TPool vs. TSession server performance with increasing requests

Figure 12 shows the result in terms of the latency of each client and server conversation as the number of client requests increased:

- Overall, as the number of requests from multiple clients increases, the latency of both TPool and TSession server increases.
- The performance of TSession server is better than TPool server when the number of client requests increases.
- When the number of client requests is fixed, the performance of TSession server is better than TPool server.

2) Performance depends on the contention of using shared data resources.

We tested the multi-threading capabilities of the ORB server when clients have conflicts accessing shared data resources. As discussed earlier, the Oracle Pro*C/C++ APIs are not thread-safe. Therefore, the shared APIs must be protected with Mutex in the contention study. All threads must obtain the lock to access the data in the database. In

constructing contention test cases, the number of contention for shared data resources increased from 1 to 25. The Request Chain Algorithm was used to send requests to the server and retrieves data from student_record table by passing in a unique student id.

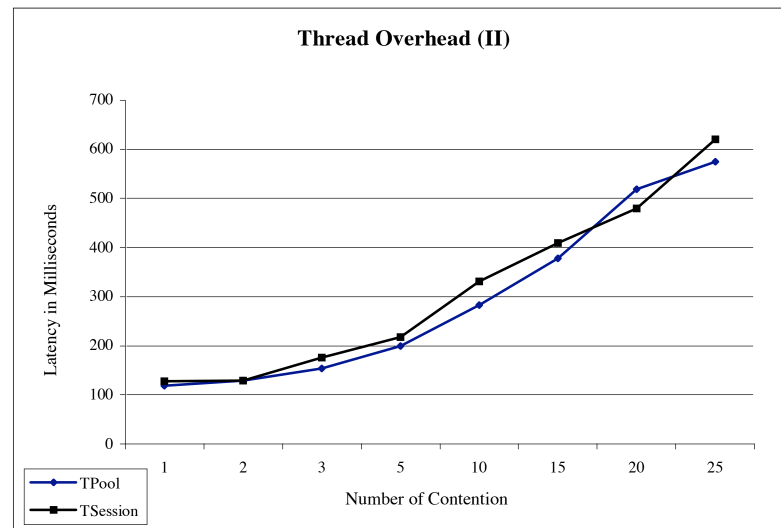


Figure 13. TPool vs. TSession server performance with increasing contention

Figure 13 shows the results in terms of latency when clients increase contention for shared data resources.

- Overall, as the number of requests from clients to shared data resources increases, the performance for both TPool server and TSession server declines.
- When the number of requests of the shared data resources increases, the synchronization overhead for both TPool and TSession server performance does not show significant differences.

The threads suffer the overhead of using locking strategy. Each thread must wait to obtain the lock to accomplish their tasks. As the number of synchronization objects increases, the performance of multi-threaded server declines, regardless of the choice between a TPool server and a TSession server.

5.3.2 Request Traffic Management

There is an advantage to use the Thread Pool model: threads are allocated based on the amount of request traffic to the server object. This means that a highly active client will be serviced by multiple threads, ensuring that the requests are quickly executed, while less active clients can share a single thread, and still have their requests immediately serviced. When using a single multi-threaded server, the Thread Pool model handles balances requests coming from multiple concurrent clients better than the Thread-per-Connection model.

We compared the multi-threading capabilities of the ORB server to evaluate efficiency between a TPool server and TSession server handling incoming requests. We ran ten concurrent clients and one server on the same machine. Each concurrent client used the Request Chain Algorithm to send different lengths of buffer characters, ranging from 1000 bytes to 4000 bytes to the multi-threaded server. The number of requests from each client varied, ranging from 1 to 20.

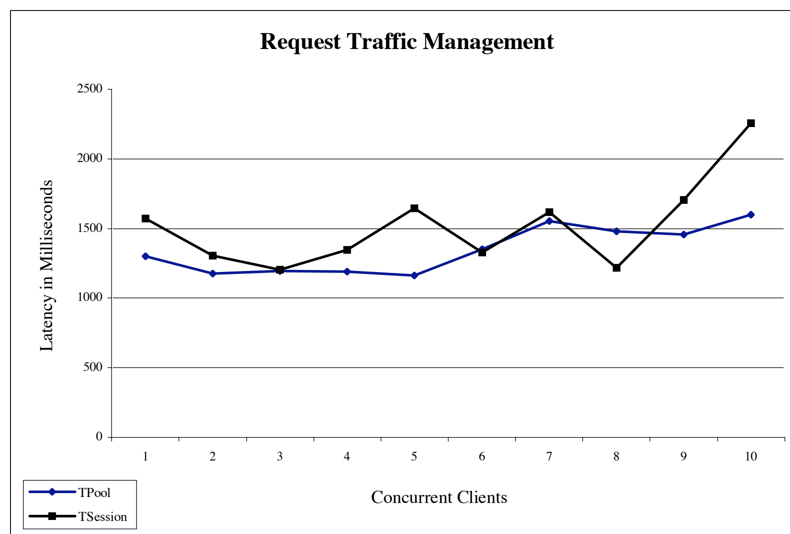


Figure 14. TPool vs. TSession server performance in handling request traffic

Figure 14 shows the result in terms of latency when using a single server to handle request traffic coming from multiple concurrent clients.

- The maximum and minimum latency using a TPool server is 1596.70 milliseconds and 1164.37
  milliseconds. The difference between maximum and minimum latency is 432.33 milliseconds.
  The maximum and minimum latency using TSession server is 2257.03 milliseconds and
  1203.92 milliseconds. The difference between the maximum and minimum latency is 1053.11
  milliseconds.
- Overall, the TPool server has better balancing than a TSession server does when handling client
  requests. In terms of requests traffic management, the performance of a TPool server is
  better than a TSession server.

In a TPool server, the requests from active clients are serviced more quickly ensures that client requests are dealt with more effectively than a TSession server.

5.3.3 Load Balancing

In Visibroker 3.3, the Smart Agent is used to locate object implementations to achieve a better load balancing of ORB server performance. If multiple servers reside on the same network, and each server implements the same objects, the Smart Agent will automatically dispatches the requests from the clients to the server with lowest load, achieving the load balancing of the server side ORB.

We tested the multi-threading capabilities of multiple ORB servers when providing load balancing. In constructing the test, we ran three servers and ten concurrent clients on the same machine. Each server implemented the same object *DataTransaction* invoked by the clients. Each concurrent client used the Request Chain Algorithm to send different lengths of characters, ranging from 1000 bytes to 4000 bytes. The test cases were the

same as in the performance study of Request Traffic Management except that we ran three servers instead of one.

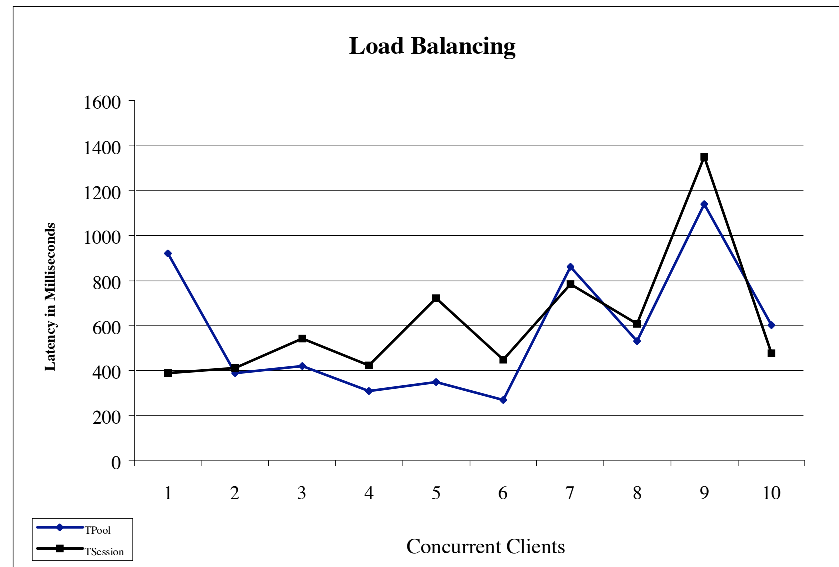


Figure 15. TPool vs. TSession multiple servers performance with load balancing

Figure 15 shows the result in terms of latency when using the multiple servers to handle request traffic coming from multiple concurrent clients. The results indicate:

- There are no significant performance differences in TPool and TSession server when load
  balancing technique being used.
- Comparing Figure 15 to the Figure 14, the performance of both TPool server and TSession server
  are better when multiple servers are employed.

When multiple servers are employed with redundantly object implementations, using the Smart Agent, the performance of both TPool and TSession increases. With more servers employed in this way, it no longer matters that which multi-threaded ORB server, TPool or TSession, is chosen because client requests will be serviced in both more quickly.

5.3.4 Object Adapter Caching

Figure 10 illustrates the work of demultiplexing client requests through several layers: network protocol stack, I/O handle, object request adapter, servant, IDL skeleton and appropriate interface operations. Layered CORBA request demultiplexing ensures more overhead, especially, when a large number of operations appear in an IDL interface or with a large number of objects managed by an ORB. One way to optimize the demultiplexing is to have the Object Adapter cache recently accessed target objects. Caching is particularly useful if client operations arrive in request trains, where a server receives a series of requests for the same target object. By caching object information, the server can reduce the overhead of locating the object for every incoming request.

We used two kinds of request invocation algorithms to determine the presence of ORB server caching. The two algorithms are Request Train Algorithm and Round Robin Algorithm. The Request Train Algorithm does not change the destination object until the client's inquiry operations are performed. The Round Robin Algorithm iterates the number of client's requests with its operation on different objects. As the multi-threaded server caches the information about recently processed objects, the Request Train Algorithm should elicit better performance than the Round Robin Algorithm.

In constructing the test cases, the number of requests was gradually increased from 10 to 100, by increment of 10. There were three objects invoked by the client: DataSend1, DataSend2 and DataSend3, as defined in the IDL file.

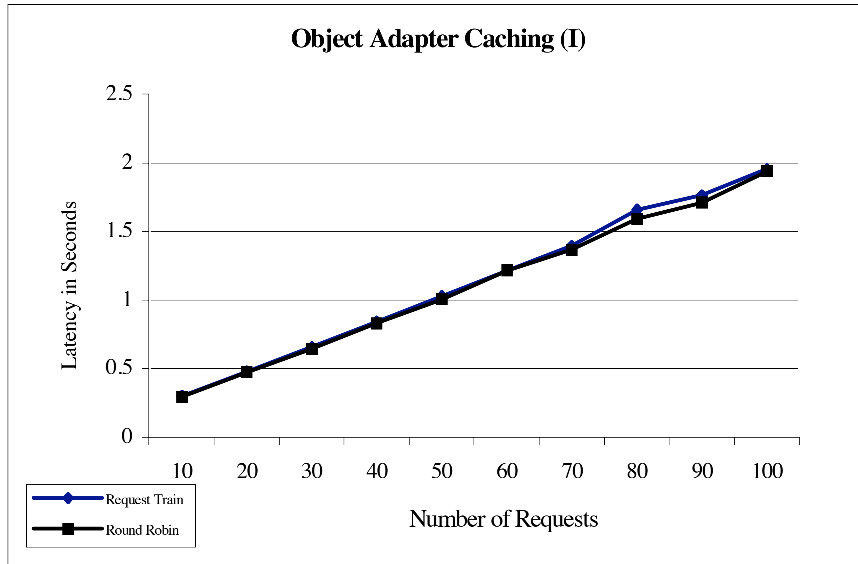


Figure 16. Request Train vs. Round Robin performance in TSession server

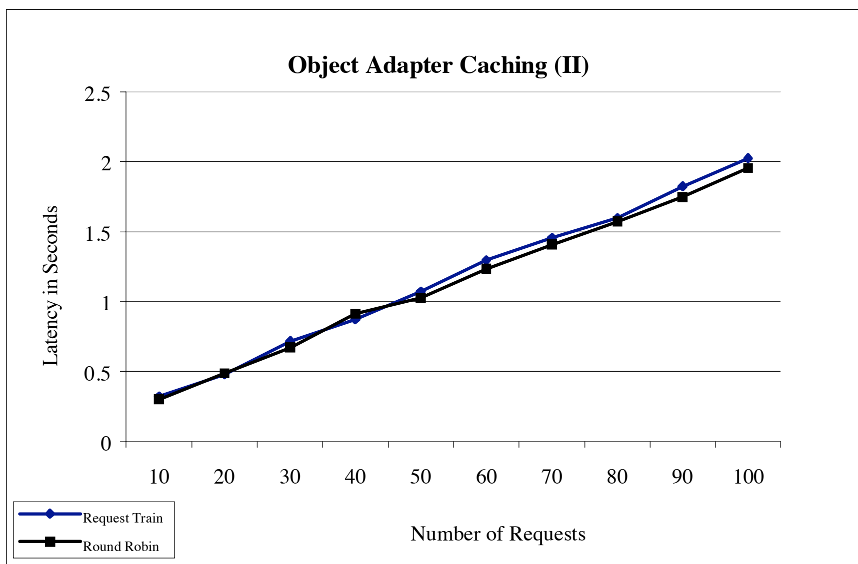


Figure 17. Request Train vs. Round Robin performance in TPool server

Figure 16 and Figure 17 show the results in terms of latency when we used two client request invocation algorithms with both the TSession server and the TPool server, the results indicate:

- With the TSession server, the performance of using Request Train Algorithm and Round Robin
  Algorithm is about the same.
- With the TPool server, the performance of using Request Train Algorithm and Round Robin
  Algorithm is about the same.

These figures reveal that the Object Adapter Caching is not supported in the current Visibroker 3.3 ORB implementation. Moreover, the multi-threading server suffers the demultiplexing overhead as well as the thread overhead. This is a disadvantage in supporting more efficient and flexible multi-threading ORB in Visibroker3.3 because demultiplexing increases the number of internal tables that must be searched as incoming client requests ascend through the processing layers of an ORB.

5.4 Conclusion

The multi-threaded ORBs implementations provide solutions in achieving performance flexibility, predictability and scalability of client/server application in a heterogeneous environment. In this report, we analyzed six major ORBs structures in detail. Overall, the analysis concludes that the Threading Framework is the most desirable architecture for multi-threaded implementations. One of the most important features of the Threading Framework is to provide capabilities to intercept or examine incoming client requests, evenly dispatch the upcall to the servers to achieve the load balancing. In addition, a thread filter can be used to stop client requests if the CPU usage of a server machine approaches to its upper limit.

We also examined the Visibroker 3.3 multi-threaded ORB implementations and evaluated Thread-per-Connection and Worker Thread Pool policies with four different

study scenarios. Multi-threaded ORB implementations, much like the other multi-threaded programs, involve a certain amount of overhead. However, the benefits of multiple threads outweigh the disadvantages. When they are used properly, multi-threaded ORB serves can handle incoming requests more efficiently. The Thread-per-Connection model is well suited for an ORB that performs long-duration conversations with multiple clients. The Worker Thread Pool model is well suited for special task handling. From a client view, it is difficulty to determine which one is better because each one has its advantages and drawbacks. Which thread policy depends on how client programs resolve thread overhead, the number of client requests, and the duration of client and server conversation.

Both the Thread-per-Connection model and the Worker Thread Pool model in Visibroker 3.3 ORB consumes a lot of demultiplexing overhead, especially when a large number of operations are in an IDL interface or a large number of ORB objects invoked by the clients. A more efficient solution to achieve high performance would be to provide caching in the Object Adapter on server side. This is not supported in the Visibroker 3.3 multi-threaded ORB implementations.

This performance study of multi-threaded ORBs has been in the environment of single processor machine. More interesting and perhaps more realistic results would be found in the multi-processor environments.

References

[1] S. Vinoski, "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997

[2] B. Nichols, D. Buttlar, and J.P.Farrell, *Pthreads Programming*, Sebastopol, CA: O'Reilly & Associates, September 1996

[3] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[4] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA*, John Wiley & Sons, New York, 1998

[5] A. Pope, *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture,* Reading, MA: Addison Wesley, March 1998

[6] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on* Programming Language Design and Implementation (PLDI), (Las Vegas, NV), ACM, June 1997

[7] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORBA*, vol. 41, October 1998.

[8] C. Hom, "The Orbix Architecture,' tech. Rep., IONA Technologies, August 1994.

[9] OrbixWeb Programmer's Guide, IONA Technologies PLC, 1997.

[10] Visibroker for C++ Programming Guide, Inprise Corporation, 1998.

[11] A. S. Gokhale and D. C. Schmidt, "Evaluating CORBA Latency and Scalability Over high-Speed ATM Networks," *in the Proceeding of ICDCS'97*, (Baltimore, MA), May 1997.

[12] G. Koch and K. Loney, *ORACLE, The Complete Reference,* McGraw-Hill, California, 1997.

[13] D. Flanagan, *Java in a Nutshell, A Desktop Quick Reference for Java Programmers*, O'Reilly Associates, CA, February 1996.

[14] J. Siegel, *CORBA Fundamentals and Programming*, John Wiley & Sons, New York, 1996.