CORBA Comparison Project

Final Project Report

Partners: MLC Systeme GmbH
Kaiserwertherstrasse 115
Rattingen, Germany

Distributed Systems Research Group
Department of Software Engineering
Faculty of Mathematics and Physics
Charles University
Malostranske namesti 25
Prague, Czech Republic

Date: June 17 1998

Pages: 88

1 Table of Contents

2 Introduction

In 1990, the Object Management Group (an international consortium now consisting of over 800 companies) has introduced its Object Management Architecture (OMA), a standardized architecture for distributed computing in a heterogeneous environment. In OMA, a pivotal role is played by the Object Request Broker (ORB), specified by the Common ORB Architecture (CORBA).

Over the years, both the CORBA standard (whose current version is CORBA/IIOP 2.2) and the implementations of CORBA have evolved considerably. Right now, major CORBA vendors offer a range of C++ and Java ORBs, that (while usually still CORBA 2.0 compliant) differ in many significant aspects. For a customer who is not intimate with the many OMG standards, it is becoming increasingly difficult to assess the practical impact of the differences between various ORB implementations.

2.1 Project Objectives

It is a goal of this project to identify a set of criteria that would help a customer to evaluate an ORB implementation, and to evaluate a set of ORBs using these criteria. The criteria cover those aspects of the ORB functionality that are important from the application developer's point of view (this includes functionality specified in the OMG standards, together with useful proprietary extensions). The impact of each criterion is discussed with respect to different ORB usage scenarios. For measurement and testing purposes, a suite of benchmarking applications is developed.

2.2 Involved Parties

The CORBA comparison project is done for MLC Systeme GmbH, Ratingen, Germany (http://www.mlc.de) by the Distributed Systems Research Group at the Department of Software Engineering, Charles University, Prague, Czech Republic (http://nenya.ms.mff.cuni.cz).

2.3 Document Organization

The document starts with a brief overview of the major OMA architecture features, putting a special emphasis on the Object Request Broker (CORBA) and Object Services (CORBAservices) standards. Following the overview, chapter 4 identifies important evaluation points with respect to functionality, performance, code portability, and application interoperability, as these are the major topics an application developer will most likely be interested in.

The core part of the report is organized using a template concept. In chapter 5, a generic template for ORB comparison is presented, with the individual comparison criteria listed and explained in detail (the criteria serve to evaluate the points identified in chapter 4). Then, each of the subsequent chapters focuses on a single ORB (omniORB in chapter 6, Orbix in chapter 7, VisiBroker in chapter 8), using the template introduced in chapter 5 to evaluate the ORB. In chapter 9, the results of the individual measurements are compared and summarized.

3 Overview

3.1 Object Management Architecture

The Object Management Group, Inc. (OMG), founded in 1989, is an international organization grouping system vendors, software developers, and end-users. The main goal of OMG is to promote object-oriented technology by defining "*a living, evolving standard with realized parts, so that application developers can deliver their applications with off-the-shelf components for common facilities like object storage, class structure, peripheral interface, user interface, etc.*" [OMA]. To achieve this goal, OMG has defined the Object Management Architecture (OMA) characterized by definition of two models: the *Core Object Model* (and its extensions called *profiles*) and the *OMA Reference Model*.

OMG's understanding of the object-oriented paradigm is reflected in the Core Object Model. The model defines such concepts as object, inheritance, subtyping, operations, signatures, etc. Additional concepts can be added to create an extension (*component*). A component should not replace, duplicate, and remove concepts. Components should be orthogonal to each other. A *profile* is a combination of the Core Object Model and one or more components (typical examples of profiles are e.g. the CORBA profile, the ODP Reference Model).

With the intention to provide a broad object-oriented architectural framework for the development of object technology, OMG defined the OMA Reference Model, published in 1992. The OMA Reference Model is comprised of the following five components:

- The *Object Request Broker* component serves as a means for delivering requests and responses among objects. It is a backbone of the OMA Reference Model, interconnecting all the remaining components.
- The *Object Services* component provides a standardized functionality (defined in the form of object interfaces), e.g. for class and instance management, storage, integrity, security, query, and versioning.
- The *Common Facilities* component defines a collection of end-user facilities (such as a common mail facility) that a group of applications is likely to have in common.
- The *Domain Interfaces* component contains a set of application-domain-specific interfaces (such as a support for healthcare or electronic commerce).
- The *Application Objects* component is not standardized by OMG. Application Objects use Domain Interfaces, Common Facilities, and Object Services via the Object Request Broker.

3.1.1 Object Request Broker

CORBA was adopted by OMG in December 1991 as a joint submission of DEC, HP, etc. The first version[1] was denoted as CORBA 1.1 and was, in 1994, replaced by CORBA 1.2 which brought more or less "cosmetic changes" with respect to CORBA 1.1. A much more important development step was CORBA 2.0, which was adopted in December 1994 and which overcomes the drawbacks of CORBA 1.2 by specifying ways for interconnecting different ORBs, and also by suggesting means for establishing interoperability between CORBA-based and non-CORBA-based environments, such as Microsoft's COM/OLE. CORBA 2.1 followed in August 1997, CORBA 2.2 (which is the current standard) in February 1998. Throughout the paper, we will mostly talk about CORBA 2.0 and 2.1, because these are the versions of the standard that are most widely available (CORBA 2.2 is still not implemented by most vendors).

[1] CORBA 1.0 was more of a working draft than a usable specification

The structure of a CORBA ORB is depicted on figure 1. The OMG standard defines the structure in terms of interfaces and their semantics. In an actual implementation, however, there is no obligation to reflect the structure; it is only necessary to preserve the interfaces and their semantics.

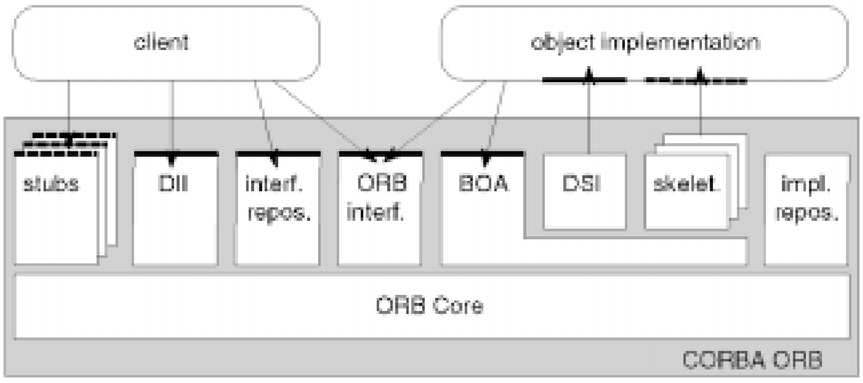


Fig. 1: Structure of the CORBA ORB

3.1.1.1 Client Side (Stubs, DII, Interface Repository)

To request a service from an object, the client can use either static or dynamic invocation. The static invocation mechanism (SII) requires the client to know the IDL definition of the requested service at compile time. The definition is used to automatically generate *stubs* for all requestable operations. To request an operation, the client calls the appropriate stub which passes the request to the ORB Core for delivery.

As opposed to static invocation, the dynamic invocation mechanism requires the client to know the IDL definition of a requested service at run time. To request an operation, the client specifies the target object, the name of the operation, and the parameters of the operation via a series of calls to the standardized *Dynamic Invocation Interface* (DII). Again, the request is then passed to the ORB Core for delivery. The semantics of the operation remains the same regardless of the invocation mechanism used. To obtain an IDL interface definition at run time, the client can use the *Interface Repository*. The repository makes IDL definitions (e.g. module, interface, constant and type definitions) available in form of persistent objects accessible via standardized interfaces. The ORB is responsible for finding the target implementation, preparing it to receive the request, and communicating the data of the request. The client need not care about the location of the object, language of the implementation, or other things not described in the interface.

On the client side, a target object is usually represented as a proxy (object) [Sha86] supporting the same interface as the target object. In the case of static invocation of an operation `m`, the proxy's method `m` calls the stub associated with `m`; the stub is responsible for creating the request and passing it to the ORB Core for delivery to the target object. In case of dynamic invocation, the client's code creates the request dynamically by calling the DII interface; in principle, no proxy and stub are necessary.

3.1.1.2 Implementation Side (Skeletons, DSI, Implementation Repository, Object Adapters)

Again, either the static or the dynamic mechanism can be used to convey the request to the actual service implementation. The static mechanism (SSI) requires the IDL definition of the requested service to be available at compile time. An automatically generated operation-specific *skeleton* is then used by the ORB to call the implementation of the requested service.

The dynamic request delivery mechanism expects the implementation to conform to the standardized *Dynamic Skeleton Interface* (DSI). Through this interface, the ORB dispatches the request to the target object (to the service implementation). In analogy to the dynamic invocation mechanism on the client side, no compile time information about the interfaces is needed. The information necessary to locate and activate implementations of requested services is stored in the *Implementation Repository*. The implementation repository is specific to a particular ORB and environment and is not standardized by CORBA.

To access services provided by the ORB itself, an implementation uses the *Object Adapter*. To satisfy the needs of diverse environments, the ORB can be equipped with several different object adapters. Each ORB, however, must provide the standardized *Object Adapter* interface—*Basic Object Adapter* (BOA) for pre-CORBA 2.2 ORBs and *Portable Object Adapter* (POA) for CORBA 2.2 ORBs. The operations of the *Object Adapter* include generation and interpretation of object references, authentication of clients, and activation and deactivation of target objects.

3.1.1.3 ORB Interface

For accessing the general services provided by an ORB, both client and implementation use the *ORB* interface. The operations of this interface include converting object references to strings and vice-versa, determining the object implementation and interface, duplicating and releasing copies of object references, testing object references for equivalence, and ORB initialization.

3.1.1.4 ORB Core, Interoperability

The initial CORBA standard left the implementation of the ORB Core totally to the ORB vendor. It was hence almost impossible to interconnect two or more CORBA-compliant ORBs released by different vendors. This lack is remedied by the CORBA 2.0 specification that defines two approaches for achieving interoperability: the *General Inter-ORB Protocol* (GIOP) and *Inter-ORB bridging*.

The General Inter-ORB Protocol specifies a set of low-level data representations and message formats for communication between ORBs. It is designed to be simple enough to work on top of any underlying connection-oriented transport protocol, such as TCP/IP and IPX/SPX. The GIOP based on TCP/IP is called the *Internet Inter-ORB Protocol* (IIOP). CORBA 2.0 requires GIOP as the mandatory protocol. To allow "out of the box" interoperability, CORBA 2.0 specifies the optional *Environment Specific Inter-ORB Protocol* (ESIOP) as an alternative to the GIOP. It has been mainly specified to allow integration of DCE with CORBA (DCE/ESIOP).

The main idea of bridging lies in mapping requests from one vendor's format to another when crossing ORB boundaries. Bridging can be performed either at the ORB Core level as an ad hoc solution for every ORB_i-ORB_j pair (*in-line bridging*), or at the application level (*request-level bridging*). Request-level bridges mediate requests by utilizing DII and DSI mechanisms to dynamically create and dispatch requests. Bridges are currently mainly used for connecting CORBA with non CORBA-compliant platforms, such as COM/OLE.

3.1.2 Object Services

3.1.2.1 Naming Service

The purpose of the Naming Service is to provide applications with means to locate objects in a network. With the help of the Naming Service, applications can assign hierarchical names to objects and look up objects using thus assigned names.

3.1.2.1.1 Typical Usage

Typically, the Naming Service is used to obtain initial access to prominent server objects, such as factories. When launched, a server explicitly assigns well known names to the objects that are to be used by clients to obtain access to the server. To access a server, a client then queries the Naming Service using the well-known name and uses the objects thus obtained.

3.1.2.1.2 The Standard

The OMG Naming Service Standard [OMG93C] defines the general format of names, the naming contexts that allow names to be grouped within a context hierarchy, and the operations to manipulate (create, bind, look up, delete) names and contexts. The standard also defines a special set of operations for lightweight name manipulation in a form of an optional names library.

The standard defines a simple name as a pair of strings, denoted as the identifier string and the kind string. When an object is assigned a name, the identifier string should describe its identity and the kind string its type (this is analogous to file name suffixes in Unix or file name extensions in DOS).

Simple names can be grouped within a context (again, this is analogous to the directory concept of Unix or DOS). Contexts can also refer to other contexts, thus creating a naming context hierarchy. Compound names can be used to navigate through the hierarchy (analogous to paths in Unix or DOS).

The standard also defines an optional API, called the Names Library. The purpose of the Names Library is to provide an easy-to-use interface for manipulating name objects handled by the service. The Names Library is not vital to the Naming Service, all functions provided by the service can be used without the Naming Library.

3.1.2.2 Events Service

The purpose of the Events Service is to provide applications with a loosely-coupled communication mechanism. Using the Events Service, applications can broadcast and receive asynchronous messages using a number of different models (e.g. push or pull model, untyped or typed events).

3.1.2.2.1 Typical Usage

In a typical Events Service usage scenario, event channels are created and registered with the Naming Service for individual event types or for classes of related event types. Applications interested in broadcasting or receiving events register themselves with the channels as event suppliers or consumers, the channel then takes care of delivering events.

3.1.2.2.2 The Standard

The OMG Events Service Standard [OMG93A] defines the roles of event suppliers and event consumers for objects that generate and process events, respectively. Two communication models are defined depending upon who is active in the communication process—in the push model, a supplier calls its consumer to deliver data, whereas in the pull model, a consumer calls its supplier to request data.

Although principally possible, neither supplier nor consumer is expected to call the other communicating object directly. Instead, all events are passed through an event channel. From the supplier's point of view, the event channel acts as a consumer; from the consumer's point of view, the event channel acts as a supplier. The event channel also provides operations to register both suppliers and consumers. Subject to the actual quality of service, the event channel can provide one-to-one, one-to-many, or many-to-many communication, and other additional features.

Communication among objects can be either untyped or typed. Untyped communication relies on suppliers and consumers having a standardized interface capable of passing an object of the class any as an event data. With typed communication, suppliers and consumers are expected to agree on a proprietary oneway operation to pass event data in a suitable format.

3.1.2.3 Trading Service

The purpose of the Trading Service is to provide applications with means to locate a service that meets specific criteria. With the help of the Trading Service, servers can describe themselves using a set of properties. Clients can then use the Trading Service to locate a server with specific properties.

3.1.2.3.1 Typical Usage

In a typical scenario, a server gives the Trading Service a description of a service and the location of an interface where that service is available. To locate the service, a client asks the Trading Service for a service having certain characteristics. The Trading Service checks against the service descriptions it holds and responds to the client with the location of the selected service's interface.

3.1.2.3.2 The Standard

The OMG Trading Service Standard [OMG96G] defines the Trader interface (Trader is an object through which the Trading Service is accessed), the Property objects used to describe a service (a Property object is a named any value), and the Standard Constraint Language to be used by Trader clients to formulate a service query.

Apart from specifying the core functionality of the Trader, the OMG Trading Service Standard also defines a mechanism through which different traders can interoperate, creating a federation of traders capable of jointly responding to a client query. The mechanism supports defining explicit link among traders and providing proxy offers.

3.1.2.4 Transaction Service

The purpose of the Transaction Service is to provide applications with a support for ACID (atomic, consistent, isolated, durable) transactions. With the help of the Transaction Service, clients can encapsulate individual operations within a transaction (provided the objects these operations are invoked upon cooperate with the Transaction Service).

3.1.2.4.1 Typical Usage

In a typical scenario, the Transaction Service client begins a transaction by issuing an explicit request to an object defined by the service. The client then issues requests, these are implicitly associated with the transaction. The client ends the transaction by issuing another explicit request to the service, depending upon circumstances the transaction is then either committed or rolled back.

3.1.2.4.2 The Standard

The OMG Transaction Service Standard [OMG94A] defines the roles of individual objects within a transaction, together with the scenario of their interaction and the necessary interfaces that ensure the client and server code portability.

Objects that can participate in a transaction are called transactional objects. A special case of a transactional object is a recoverable object, by definition an object whose state is directly influenced by transaction commit or rollback. Recoverable objects cooperate with OTS by registering resources during transaction, OTS will then synchronize the commit and rollback operations upon registered resources. Finally, a recent OTS update introduces the Synchronization interface used to notify objects prior to transaction commit.

The Transaction Service does not deal with locking of resources. For this purpose, OMG provides the Concurrency Control Service [OMG94E]. The service implements an interface for hierarchical locking of objects.

3.1.2.5 Persistence Service

The purpose of the Persistence Service is to equip servers with means to provide persistency for objects. Relying on the Persistence Service, servers can provide objects whose lifetime is not limited by the server execution time.

3.1.2.5.1 Typical Usage

Typically, the support for persistence includes means to preserve the state of implementation objects and means to activate and deactivate the implementation objects as needed. This involves a varying level of cooperation on the server implementor side.

3.1.2.5.2 The Standard

With the OMG Persistent Object Service standard [OMG94D] being phased out of the CORBA environment and its successor still in design phase (a deadline for accepting revised submissions to the Persistent State Service RFP ends May 1998), no relevant OMG standard exists.

3.1.2.6 Lifecycle Service

The purpose of the Lifecycle Service is to provide applications with basic operations related to object lifecycle. The lifecycle operations include creating, deleting, copying and moving of objects.

3.1.2.6.1 Typical Usage

Typically, objects are created using object factories (an object factory is simply an object used to create other objects) and deleted either explicitly, or automatically using a reference counting mechanism. Operations to copy and move objects are not provided by typical ORB implementations.

3.1.2.6.2 The Standard

The OMG Lifecycle Standard [OMG93B] defines the role of object factories together with an interface through which the object deletion, copy and move operations are provided. Unfortunately, the text of the standard describes little of how the service is to be implemented. Hence, except for some of the high-level concepts, the standard is rarely (if ever) used.

4 Criteria Discussion

The evaluation criteria are separated along the same lines as is the Object Management Architecture—the individual sets of criteria focus on the Object Request Broker and the Object Services (the report does not evaluate other parts of the OMA architecture). In addition to that, the criteria also list points of general interest to an application developer, such as the quality of documentation, development support, reliability.

4.1 Object Request Broker

4.1.1 Standardized Functionality

Of the entire OMA architecture, an application developer will probably be most interested in the properties of the ORB. The basic criteria related to the ORB functionality, as described in the CORBA standard, include:

- Adherence to the OMG Interface Definition Language (IDL) specification and the appropriate language mapping.
- Basic remote invocation functionality (e.g. support for remote calls, exceptions, contexts, the functionality of the ORB and object adapter (BOA or POA) interfaces).
- The interfaces for manipulation with dynamically defined types (the Dynamic Invocation Interface (DII) on the client side, the Dynamic Skeleton Interface (DSI) on the server side, CORBA 2.2 ORBs should also provide the `DynAny` data type).
- Functionality of the Interface Repository (IR) and the Implementation Repository.
- Interoperability.

4.1.2 Nonstandard Extensions

Due to the incremental nature of OMG standards' evolution, ORB vendors were forced to offer proprietary solutions to many open problems in their early ORB implementations. For backward compatibility, cost efficiency and other reasons, some of the proprietary solutions persist (as do some of the open problems related to ORB implementation). Often, the proprietary mechanisms provide important functionality, and thus cannot be overlooked simply for being proprietary.

4.1.2.1 Communication Extensions

4.1.2.1.1 Locating Objects

One of the very basic features required by ORB clients is the ability to locate an object given a set of criteria (e.g. server host, interface). In the standardized OMA architecture, this feature is to be provided by the Trading Service (see section 3.1.2.3), but since this service was standardized relatively late (August 1996, which is 5 years after the first CORBA standard appeared), many vendors have introduced proprietary solutions in the meantime.

4.1.2.1.2 Binding To Objects

The standard way to bind to an object is to get a bootstrap reference to the Naming or Trading Service and to use that reference to obtain whatever other references are necessary. Alternatively, the `string_to_object()` operation of the ORB can be used to import stringified object references. In addition to (or in place of) these functions, the ORB vendors often introduce proprietary binding mechanisms.

4.1.2.1.3 Instantiating Implementations

The instantiation of a CORBA object on the server side of an application consists of

(1) creating of the actual implementation object

(2) creating of the unique object reference

(3) registering of the implementation object with the reference

The mechanism to do this was described poorly in the original CORBA standard (this is fixed in CORBA 2.2) and thus its implementations differ in the individual ORBs. The same goes for the object deletion mechanism.

4.1.2.1.4 Persistent References

An important feature of the ORB is the support for persistent object references. These are necessary when implementing persistent objects and are usually tied to some sort of a server activation mechanism. Again, the original CORBA standard is not detailed enough in this area (fixed in CORBA 2.2) and thus most ORBs offer proprietary solutions.

4.1.2.1.5 Automatic Activation

An important part of the ORB functionality, used especially with persistent servers and servers that are to provide a higher degree of reliability, is the ability to launch a server on demand.

4.1.2.1.6 Automatic Reconnection

Related to the automatic activation mechanism is a mechanism that allows a client to automatically reconnect to a server in case of a communication failure.

4.1.2.1.7 Invocation Extensions

Extensions to the standard invocation mechanism (e.g. asynchronous invocations, callbacks, proxy objects modified by programmer) are provided by most ORBs as well.

4.1.2.1.8 Customizability

Last but not least, it is important to have the possibility to customize the ORB behavior (e.g. intercepting and filtering of messages, piggybacking of data onto messages, event hooks).

4.1.2.2 Multi-threading Extensions

4.1.2.2.1 Multi-threaded Servers

The most common case of a multi-threading extension is a support for multi-threaded server implementations. Here, several basic strategies of server implementation behavior appear:

- *single thread* - all incoming requests are serialized and processed by a single thread.
- *thread per request* - for every incoming request a new thread is created. After processing a request, the thread terminates.

- *pool of threads* - during the start of the server, a pool of threads is created to handle incoming requests. Each thread waits for an incoming request, handles it and loops back to repeat this sequence. This strategy allows for parallel processing, while limiting the server's use of resources.

- *thread per client* - a separate thread is created for each client process that connects to a server. This could be useful e.g. when consecutive requests from the same client form a single transaction. A simple modification of this strategy is thread per client connection.

- *thread per object* - a separate thread is created to handle requests for one server object only (or for a subset of server objects). This could be useful e.g. in real-time processing, where the thread associated with one object has higher priority than the others.

Clearly, it is helpful when the ORB supports at least some of the strategies listed above. It is also important that the ORB is flexible enough to allow other (proprietary) strategies to be implemented.

4.1.2.2.2 Multi-threaded Clients

Apart from employing multi-threaded servers, there are at least two strong benefits from employing multi-threaded clients:

- *non-blocking call* - a special thread is created to make a (time consuming) remote operation. The client can continue processing without waiting for the result of the operation. Similar semantics can be achieved using (several) `oneway` or `deferred` operations.

- *call-back receive* - a special thread can be created to handle all incoming call-backs without having to poll for communication events.

4.1.2.2.3 Multi-threaded ORB

Finally, the issue of employing multi-threading inside the underlying ORB is also important. Here, two questions arise:

- does an ORB employ threads to process its internal actions ?

- what is an ORB internal policy for managing TCP/IP connections between clients and servers ?

For client or server programmers, a possibility to setup (or change the default) internal ORB behavior is also important.

4.1.2.2.4 Concurrency

Closely related to the multi-threading is the issue of concurrency control. The shared data structures have to be protected to avoid corruption by parallel processing. This protection has to apply both at the ORB level and at the application level.

4.1.2.3 Management Extensions

In most cases, the installation of an ORB running on a single computer (or in a relatively small network) is simple to configure and manage. With the growing number of network hosts running ORB clients and servers, however, the management becomes complicated and a need for a sophisticated configuration and management tool arises. As there are no OMG standards for ORB configuration and management, the configuration and management tools (when provided) are proprietary and offer functions specific to a given ORB.

4.1.3 Performance Criteria

With the purpose of the ORB being the delivery of requests to objects, the performance criteria are (naturally) related to the request delivery mechanism. Since the exact character of an object invocation may differ (e.g. using SII/DII, SSI/DSI, objects collocated/distributed), the number of combinations is too large to be covered in this paper. Thus, we have decided to only cover what we see as the most common usage case, that is the static request delivery mechanism run over a network (most of the notes are relevant to the other cases as well).

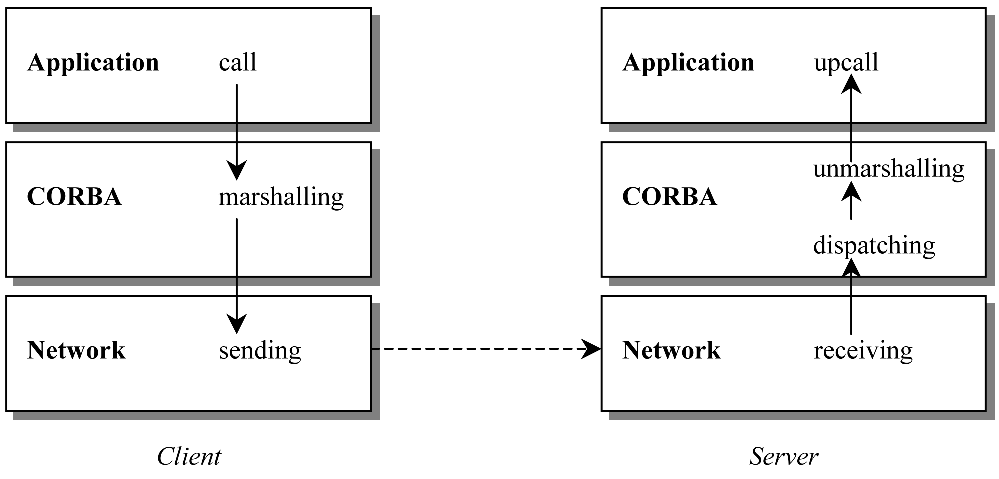


Fig. 2: Simplified CORBA invocation path

When a CORBA client invokes an operation upon a server, the call goes through several phases:

- A call to the object proxy.

  Typically, this is a standard language call and does not differ from what a local call to the same object would look like. The proxy object is responsible for communicating the request to the actual object implementation.

- The proxy marshals the client's request.

  Marshalling is a translation of the procedure call to a network message, consisting of initializing the request and filling in the object name, operation name, and the arguments of the call. In general, two of the most important performance issues related to marshalling are the speed of marshalling and the efficiency of the encoding of marshaled data. As the encoding of the marshaled data is standardized (IIOP mandates use of the CDR format for data representation), it is the speed of marshalling that is the most important comparison criterion here.

  The speed of marshalling depends on the basic nature of the ORB implementation (e.g. management of network buffers, avoidance of excessive data copying) on the data types that are marshaled (e.g. some types may be more expensive to marshal than others). While the first factor is generally exhibited by all calls, and is especially visible on calls that transfer a small number of simple arguments, the second factor can be evaluated by comparing the invocation times of calls with different argument types.

- The marshaled form of the request is sent over the network.

  With most ORBs today being built atop TCP/IP, the performance related to the network traffic is mostly determined by the quality of the TCP/IP protocol stack. As the amount of data passed as a part of the request is (with a negligible number of exceptions, such as the object IDs) standardized, the ORB can only influence the speed of the data transfer in a limited number of ways (e.g. by carefully arranging the calls to the TCP/IP stack, by setting the parameters of the transport protocols). Unfortunately, these different individual approaches are difficult to compare, because their impact varies with many circumstances.

In addition to the basic communication mechanism built using IIOP over the TCP/IP protocol stack, some of the ORBs also provide further communications extensions. Among these, most interesting is the local optimization of the communication protocol. As the purpose of the optimized local communication mechanism is usually to improve the ORB's performance by bypassing the TCP/IP protocol stack (and certain parts of the marshalling as well), the simplest way to evaluate the mechanism is by comparing its performance with that of the basic network communication mechanism provided by the ORB.

- On arrival to the server, the request is dispatched by the ORB.

  Here, dispatching means locating the object implementation the request should be passed to (and what operation should be invoked). This decision is made based on the object reference and the operation name, both are enclosed in the request packet.

  Strictly speaking, the process of dispatching really begins within the network layer of the operating system (or, in some cases, even within the network hardware), the ORB on the server side of a distributed application is usually involved only after the IIOP message is delivered to it by the network protocol stack. The ORB needs to decide what object instance to contact and what code to use for unmarshalling of the request arguments (since this implies searching, scalability is also an issue).

- Unmarshalling the request packet.

  As the name indicates, unmarshalling is a process opposite to marshalling—the arguments of the call are converted from the encoded network request form to the data types that the object implementation understands.

  In many respects, unmarshalling is similar to marshalling. The important differences are

  - The memory allocation (usually, ORB needs to allocate memory for the unmarshalled data)
  - The unmarshalling of object references (when unmarshalling an object reference, the table of objects present in the target address space is searched, or a proxy is created, or both, which again raises the issue of scalability).

- Upcall to the object implementation.

The steps taken during the delivery of the call results from the server back to the client are very similar to those just described. The reply from the server is marshaled and sent across the network; on the client side it is received, unmarshaled and delivered to the client code. The difference here is that there is no dispatching on the client side (except for locating an outstanding request that matches the reply for ORBs that are capable of serving multiple requests over the same connection).

To summarize, the most important features of the ORB with respect to the performance of the request delivery mechanism are the speed of marshalling, the efficiency of the communication mechanism, the speed of dispatching (especially with respect to scalability), and the speed of unmarshaling.

4.1.4 Benchmarking

As is clear from the previous section, the performance of an ORB is influenced by a number of more or less independent factors. Although this influence can often be measured (by hooks within the ORB code or the operating systems primitives), it is not trivial to combine the results of the individual measurements into a comprehensive picture of the ORB performance. On the other hand, the measurements that evaluate the overall ORB performance (e.g. end-to-end performance, roundtrip times) are not enough to characterize the influence of the individual environment features.

Roughly speaking, the first kind of measurements (focus on details) is useful mostly for ORB developers, while the second kind of measurements (focus on overall results) gives an overview of the ORB performance in a specific environment to application programmers.

As the purpose of this report is to provide information for application developers, we have decided to orient most of the benchmarks as end-to-end. However, the end-to-end performance benchmarks alone would only be relevant for a small set of applications (those that run in the same environment as our tests and that use the ORB in a similar manner). Thus, we try to illustrate the dependency of the results on the environment with additional measurements. As a result, we hope to obtain a set of benchmark measurements that characterize an ORB in a realistic environment and that can be used to judge on the behavior of an ORB even in an environments that differs from the one that was used to do the benchmarks.

The individual benchmarks described in this report are (for a detailed description, see the template chapter 5):

- *CALL* – evaluates the behavior of the dispatcher, verifies the correctness of the benchmarks with simple invocation patterns.
- *IDL.DEEP, IDL.LONG, IDL.NAMES* – evaluate the dispatching ability of the ORB, also serve to discover some of the limitations of the IDL compiler with respect to the structural complexity of the IDL source.
- *IDL.ENCAP, THROUGH.IN* – evaluate the speed of the ORB when passing different data types and different amounts of data in the direction from client to server.
- *PROXY, PROXY.LOT* –evaluate the scalability of both the client and the server with respect to the number of object implementations or proxy objects, also discover scalability limitations and behavior of the proxy creation operations.
- *ONEWAY* – evaluates robustness of the server with respect to the number of pending request, also tests the behavior of oneway calls.
- *MT.CLIENT, MT.CONN, MT.POOL* – evaluate the behavior of the support for multi-threaded clients and servers.

4.2 Object Services

The spectrum of the Object Services is too large to be covered in sufficient detail here. Hence, we only limit ourselves to summarizing the major features of the services offered by individual ORB vendors, with more detailed attention paid to the database and transactions area, as requested by MLC Systeme GmbH. For a detailed discussion of some of the services, see e.g. [Bal96A], [Har97A], [Hri96A], [Kle96B], [Kle96C], [OMG97J], [Tum97A].

4.3 Miscellanea

Apart from the criteria specific to the Object Management Architecture and CORBA in particular, we also consider the following points important for application developers:

- Technical requirements (e.g. supported platforms, compilers, resource consumption).
- Support for developers (e.g. quality of documentation, technical support, debugging features).
- Vendor characteristics (e.g. expertise, pricing policy).

5 Evaluation Criteria

Broker:	**Template**
Vendor:	**N/A**
Version tested:	**N/A**
Platforms used:	**N/A**

This is a template chapter, describing the individual comparison criteria used to evaluate the various object request broker implementations. A brief overview of the ORB is presented in the first paragraph of each specific chapter.

5.1 Object Request Broker

In this section, the ORB's compliance to OMG standards is evaluated. Nowadays, most ORB vendors have no problems coping with the core of the CORBA 2.0 specification, but certain (usually recent) additions to the standard might not be implemented. Importance of this criterion depends on whether the customer minds tying himself to a specific ORB implementation (in most cases, ORBs provide proprietary functionality as a substitute for the missing standardized parts).

5.1.1 Standardized Functionality

5.1.1.1 Interface Definition Language

This section focuses on the support for the Interface Definition Language within the ORB, including its mapping to the implementation language (C++ in our cases). Some of the limitations of the IDL compilers with respect to the structural complexity of the IDL sources are mentioned as well (these are obtained using the IDL.NAMES, IDL.LONG and IDL.DEEP benchmarks, see section 5.1.1.2.2).

5.1.1.2 Basic Remote Invocation

This section focuses on the basic remote invocation functionality of the ORB (e.g. support for remote calls, exceptions, contexts, object adapter functionality). The section also includes the following three groups of benchmarks:

5.1.1.2.1 Measurement Verification

To verify the validity of the benchmark results, it is important to test whether the areas in which the benchmarks are over-simplified (e.g. calling the same object over and over again, calling the object in predictable patterns) do not influence the results. Thus, we have devised a simple suite of benchmarks that tests the dependence of the invocation times on the invocation strategy used in the benchmark.

CALL

The benchmark employs the following five invocation strategies (regardless of strategy, the total number of objects and calls used in the benchmark remains constant):

- *As Thru, Rev Thru* – a single object is called 10 times, this is repeated for 1000 objects in their creation order and reverse creation order (this is similar to the behavior of e.g. the *THROUGH.IN* benchmark).
- *As Proxy, Rev Proxy* – 1000 objects are called once in their creation order and reverse creation order, this is repeated 10 times (this is similar to the behavior of e.g. the *PROXY* benchmark).

- *Random* – 10000 calls are made to 1000 objects, targets are randomly selected.

The results of the benchmark show the influence of the invocation pattern on the results.

5.1.1.2.2 Dispatcher Performance

The nature of the IDL-to-C++ mapping and of the IIOP protocol implies dependencies between the structural complexity of the IDL interfaces and the performance of the applications based on these interfaces. The exact nature of this relationship depends mostly on the quality of the ORB dispatcher code (partially a part of the object adapter, partially produced by the IDL compiler).

The *IDL.NAMES*, *IDL.LONG* and *IDL.DEEP* benchmarks are aimed at discovering weaknesses within the dispatcher code, especially with respect to the complexity of dispatching algorithms (e.g. linear searches through a potentially large amount of data).

IDL.NAMES

In the *IDL.NAMES* benchmark, a single IDL interface within a single IDL module is defined. The interface defines several operations that differ in the length of their name, in a single character at the beginning of the operation name or at the end of the operation name, in the length of the argument name:

```
module test
{
	interface iface
	{
		double x ();
		double x2345<500chars>7890 ();
		double xAAAA<500chars>AAAA ();
		double yAAAA<500chars>AAAA ();
		double AAAAA<500chars>AAAx ();
		double AAAAA<500chars>AAAy ();
		double shorta (in long a, in long b);
		double longa (in long a<500chars>, in long b);
	};
};
```

The round trip time (RTT) needed to return from an invocation of the operations is measured.

IDL.LONG

A single IDL interface with a large number of operations is defined:

```
module test
{
	interface ifaceLong
	{
		double x1();
		double x2();
		...
		double x94();
		double x95();
	};
};
```

The difference in RTT when invoking operations from different positions within the interface is measured.

IDL.DEEP

Several interfaces are defined in different depths of an IDL module:

```
module test {
      interface iface1 { double x(); };
      module mod2 {
            module mod3 {
                  module mod4 {
                        module mod5 {
                              interface iface5 { double x(); };
                              module mod6 {
                                    module mod7 {
                                          module mod8 {
                                                module mod9 {
      ...
};
```

The difference in RTT between invocations of interfaces from different depths in the module is measured.

5.1.1.2.3 Performance

IDL.ENCAP

To compare the times needed to pass data individually, in an array, in a bounded sequence, and in a structure, the IDL.ENCAP benchmark was made. The benchmark compares invocation time of the following four operations:

```
typedef long Arr [100];
typedef sequence <long,100> Seq;
struct Str { long X0; ... long X99; };

interface Encap
{
      void NoEncap (in long X0, ... in long X99);
      void ArrEncap (in Arr A);
      void SeqEncap (in Seq S);
      void StrEncap (in Str S);
};
```

THROUGH.IN

To provide information on how fast does an ORB transfer data, a series of benchmarks on passing of a large amount of arguments is included. In these benchmarks, the basic data types (integers, reals, characters, octets, anys) are transferred as separate entities, within arrays, or within sequences. The arrays and sequences (bounded sequences are used) all encapsulate 1kB (1024 bytes) of data, regardless of the type transferred.

```
module test
{
      typedef float                 float_arr [256];
      typedef double                double_arr [128];
      ...
      typedef sequence<float,256>   float_seq;
      typedef sequence<double,128>  double_seq;
      ...
      typedef string<1024>          str;

      interface through {
            long infloat      (in float x);
            long indouble     (in double x);
            ...
            long floatArray   (in float_arr x);
            long doubleArray  (in double_arr x);
            ...
            long floatSeq     (in float_seq x);
            long doubleSeq    (in double_seq x);
            ...
            long StringString (in str x);
      };
};
```

The results of the benchmark show both the throughput of the ORB and the overhead associated with individual data types being passed. Both the time of delivery of the request and RTT of the request completion are measured by the benchmarks[1] (the time of delivery hints at the raw throughput of the ORB implementation, while the RTT is what interests client application programmers).

The benchmarks are collectively referred to as THROUGH.IN. The measurement results are split into groups covering e.g. the basic types, sequences, arrays.

5.1.1.3 Dynamic Type Manipulation

This section lists the ORB's support for manipulation of dynamically defined types, that is the Dynamic Invocation Interface, the Dynamic Skeleton Interface, and for CORBA 2.2 ORBs also the DynAny Interface.

5.1.1.4 Repositories

This section describes the Interface Repository and the Implementation Repository of the ORB.

5.1.1.5 Interoperability

The IIOP protocol support is evaluated in this section. We use the THROUGH.IN (see section 5.1.1.2.3) benchmark to test the interoperability of different ORBs (the test passes a wide variety of data types between the client and the server). Interoperability with non-CORBA-based environments (e.g. Microsoft COM, Java) is briefly outlined as well.

[1] The delivery time is measured in local configurations only.

5.1.2 Nonstandard Extensions

This section focuses on the nonstandard extensions provided by the ORB, as listed in section 4.1.2. The following extension groups are covered:

5.1.2.1 Communication Extensions

This section is divided into several subsections: *Locating Objects, Binding To Objects, Instantiating Implementations, Persistent References, Automatic Activation, Automatic Reconnection, Invocation Extensions, Customizability*.

5.1.2.2 Multi-threading Extensions

Here, the individual subsections discuss four features related to multi-threading: *Multi-threaded Servers*, *Multi-threaded Clients*, *Multi-threaded ORB*, *Concurrency.* We use three benchmarks to test the behavior of the multi-threading support within the ORB:

5.1.2.2.1 MT.CLIENT

The MT.CLIENT benchmark tests the multithreading capabilities of the ORB client when multiple threads access different servers. The benchmark uses five servers, each providing a synchronous operation that takes 10 seconds to complete. The client creates five threads, each of the threads invokes the operation five times on one server. The benchmark takes 50 seconds to complete when the client issues the requests from different threads in parallel, and 250 seconds when the requests are serialized.

5.1.2.2.2 MT.CONN

The MT.CONN benchmark tests the multithreading capabilities of the ORB client when multiple threads access the same server. The benchmark is similar to MT.CLIENT, but uses only one server providing five objects with a synchronous operation that takes 10 seconds to complete. The client creates five threads, each of the threads invokes the operation five times on one object. The benchmark takes 50 seconds to complete when the client issues the requests from different threads in parallel, and 250 seconds when the requests are serialized.

5.1.2.2.3 MT.POOL

The MT.POOL test does not actually measure anything. Instead, it implements the “pool of threads” server threading strategy (if possible) to test the functionality of server-side multi-threading support.

5.1.2.3 Management Extensions

The ORB configuration and management tools (if any) are described in this section.

5.1.3 Scalability

In this section, we analyze various aspects of the ORB’s scalability that were not touched by previous benchmarks. Three aspects of scalability are mentioned—speed with respect to number of objects, resource consumption with respect to number of objects, and resource consumption with respect to the number of incoming (asynchronous) requests.

5.1.3.1 Speed vs. Number of Objects

An important part of the remote invocation mechanism defined by CORBA is the ability to pass objects by reference. Whenever an ORB receives a new object reference, a proxy object is created and passed to the target application. The application can then invoke operations upon the proxy (these get forwarded to the object represented by the proxy).

Both the time needed to create the proxy object and the time needed to open a connection to the object represented by the proxy vary in different ORBs. The performance of some of the operations performed by the ORB (dispatching, object reference unmarshalling, proxy creation) depends on the number of existing objects and proxies as well.

5.1.3.1.1 PROXY

The PROXY benchmark measures the time it takes to create and return a sequence of objects and to invoke an operation on these objects (the first and all subsequent invocation times differ) This is measured for 1000, 2000 and 3000 objects on the server. The measurement results are split into three groups covering the proxy creation time (PROXY.CREATE) and the difference between the first (PROXY.FIRST) and second (PROXY.NEXT) call to a newly created proxy.

5.1.3.1.2 PROXY.LOT

The PROXY.LOT benchmark was designed to test the degradation in request delivery speed when handling a large number of objects. A client and two servers are involved in the test—the client and the first server increase the number of their objects by a thousand each test run, while the second server keeps its number of objects constantly low. A difference in invocation speed is measured for two operations:

```
interface factory {
      long nop();
      long op5(in helo h1, ... in helo h5);
};
```

5.1.3.2 Resources vs. Number of Objects

The consumption of memory per object on both the server and the client is measured using a slightly modified code of the PROXY benchmark.

5.1.3.3 Resources vs. Queued Requests

5.1.3.3.1 ONEWAY

The ONEWAY benchmark was made to test the number of oneway calls that can be issued until the client blocks. The benchmark client keeps issuing oneway void calls that take a long time to complete at the server side and measures the time it takes to issue the call. Long delays indicate the client being blocked, these are registered by the benchmark.

5.1.4 Robustness

This section is dedicated to the issue of ORB robustness. The subsections discuss the support for building reliable servers using the ORB, measure some of the limits of the ORB implementation (size of requests, number of objects). We also list some of the bugs we have found here.

5.1.4.1 Reliable Servers

The support for building reliable servers using the ORB is discussed in this section.

5.1.4.2 Limits

5.1.4.2.1 SIZE

This benchmark tries to send data packets of increasing size in a single request to find out if there are any limits imposed on the request size and to test whether the server can cope with large requests.

5.1.4.2.2 Number of Objects

The maximum number of objects the server and the client are able to cope with is tested using a slightly modified code of the PROXY benchmark.

5.1.4.3 Bugs

Some of the bugs discovered during the testing of the ORB are listed in this section.

5.2 Object Services

In this section, we evaluate the Object Services provided by the vendor of the ORB (this does not necessarily imply that another vendor's service cannot be used with the ORB though). The evaluation lists the range of services available, their degree of functionality, and the replacements for standard services. Individual subsections focus on specific services (for a description of evaluated services, see section 3.1.2).

5.3 Miscellanea

5.3.1 Requirements

This section lists the general requirements of the ORB (e.g. supported platforms, compilers).

5.3.2 Development

This section discusses development and debugging support provided by the individual ORB implementations. The focus is on support for short-term development and debugging rather than for long-term project maintenance and management.

5.3.3 Vendor

This section provides brief information on the vendor of the ORB (e.g. short company profile, pricing policy).

6 ORL omniORB

Broker:	**omniORB**
Vendor:	**Olivetti & Oracle Research Laboratory**
Version tested:	**omniORB 2.5.0**
Platforms used:	**Windows NT 4.0 Workstation**

- **200MHz AMD K5 64MB RAM**
- **166MHz Intel Pentium 64MB RAM**
- **179MHz Intel Pentium Pro 64MB RAM**

AIX 4.1 on IBM RS6000-595
network: 10BASE-T Ethernet
compilers:

- **Microsoft Visual C++ 4.2**
- **IBM C Set ++ 3**

OmniORB is a lightweight CORBA implementation—it supports the basic CORBA core (ORB and partially BOA) which is fully functional, but does not provide the Dynamic Invocation Interface (DII), the Dynamic Skeleton Interface (DSI), the Interface Repository (IR), and the Implementation Repository.

6.1 Object Request Broker

6.1.1 Standardized Functionality

6.1.1.1 Interface Definition Language

The omniORB IDL conforms to the OMG IDL specification. OmniORB only supports C++, hence only the IDL-to-C++ mapping is provided. The mapping is CORBA 2.0 conformant, IDL exceptions are mapped to C++ exceptions, IDL modules are mapped to C++ classes (instead of namespaces).

OmniORB 2.5 has problems with IDL names longer than 500 characters (the IDL compiler does not compile the IDL source). It also has problems with IDL interfaces that contain a large number of operations (the C++ sources generated by the IDL compiler do not compile under VC++ 4.2 if more than 95 operations are declared within a single interface, the limit is slightly higher for VC++ 5.0).

6.1.1.2 Basic Remote Invocation

OmniORB supports both the "at most once, synchronous" and the "best effort, asynchronous" invocations as defined by the CORBA standard. It supports exceptions. It does not support context handling.

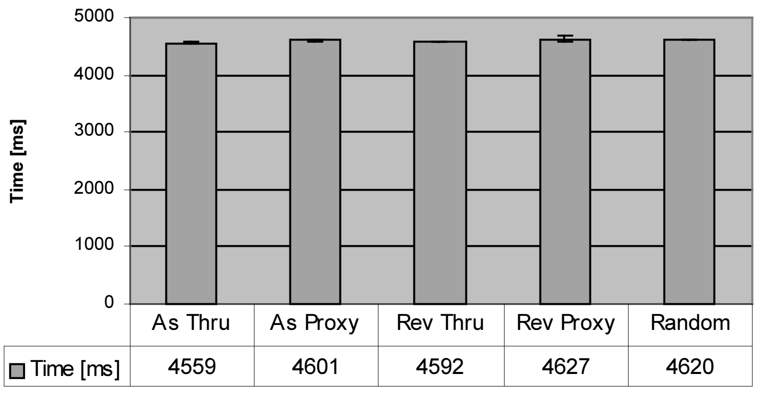
The ORB module is implemented completely according to the standard (with the exception of the calls related to DII, IR and the Implementation Repository).

The BOA module is rather incomplete—only the calls for registering object implementations (`obj_is_ready()`, `impl_is_ready()`) and for disposing object implementations (`dispose()`) are provided. The semantics of the `obj_is_ready()` call differs from the standard specification—in omniORB, `obj_is_ready()` should be called to individually register object implementations. The `dispose()` call has a modified semantics as well—it unregisters the object implementation with BOA (as specified by the standard) and deletes the object only after its reference count reaches zero. With omniORB, the `dispose()` call is the only call that should be used to delete the object implementation.

6.1.1.2.1 Measurement Verification

CALL

The results of the CALL benchmark indicate there is no significant dependence between the invocation times and the object invocation pattern. This means that the results of the other benchmarks can be applied to applications with different invocation patterns as well.



	As Thru	As Proxy	Rev Thru	Rev Proxy	Random
▨ Time [ms]	4559	4601	4592	4627	4620

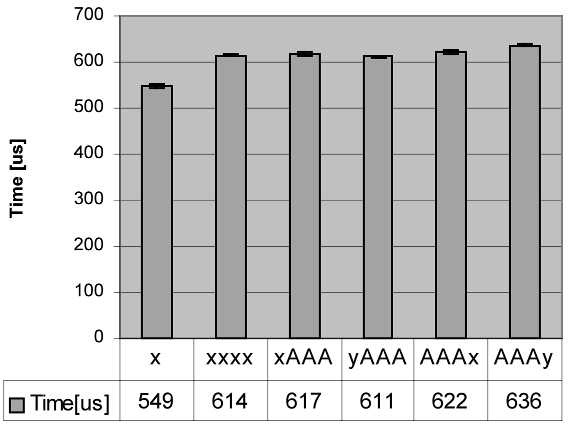
Fig. 3: *CALL* NT Int179 local (10.000 calls)

6.1.1.2.2 Dispatcher Performance

IDL.NAMES

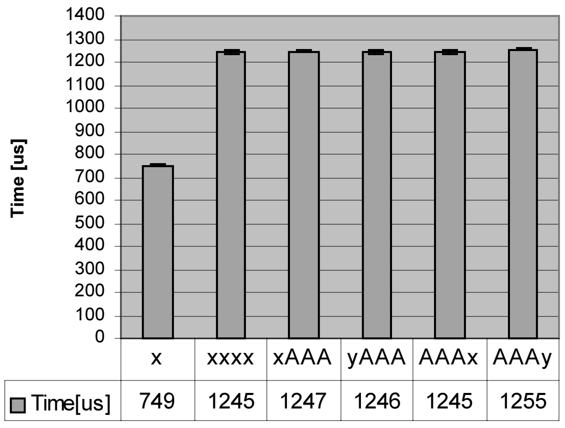
The most important result from the IDL.NAMES benchmark is the demonstration of the fact that calling operations with longer (500 characters) names takes longer time to complete. This is due to the fact that the marshalling, unmarshalling, transport and dispatching algorithms all use a plain string form of the operation names. The test shows that:

- For normal-length names (10-30 characters), the difference in invocation time is insignificant (about 1%).
- The difference in invocation time is more significant when running over a network (this is due to the fact that the operation name is transported in a plain string form).



	x	xxxx	xAAA	yAAA	AAAx	AAAy
■ Time[us]	549	614	617	611	622	636

Fig. 4: *IDL.NAMES* NT AMD200 local (1 call)



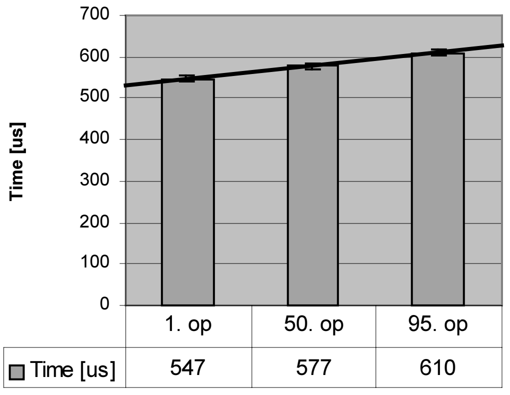
	x	xxxx	xAAA	yAAA	AAAx	AAAy
■ Time[us]	749	1245	1247	1246	1245	1255

Fig. 5: *IDL.NAMES* NT Int166 -> AMD200 idle network (1 call)

IDL.LONG

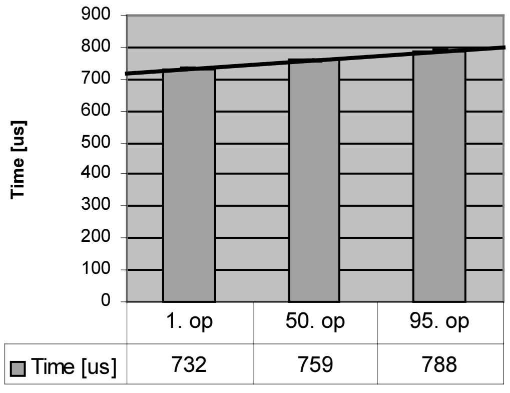
The IDL.LONG benchmark shows that the operation that is defined later in the IDL interface takes longer time to complete. The reason for this is that the dispatcher code (generated from the IDL definition) looks up the operation name using a sequential (linear) search. The test shows that:

- For medium-sized interfaces (100 operations), the difference in invocation time between the first and the last operation may be significant.
- The difference in invocation time does not change when running over a network (clearly, only server side code matters).



	1. op	50. op	95. op
■ Time [us]	547	577	610

Fig. 6: *IDL.LONG* NT AMD200 local (1 call)

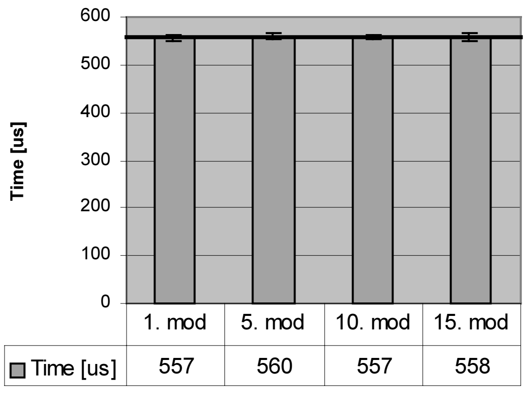


	1. op	50. op	95. op
■ Time [us]	732	759	788

Fig. 7: *IDL.LONG* NT Int166 -> AMD200 idle net (1 call)

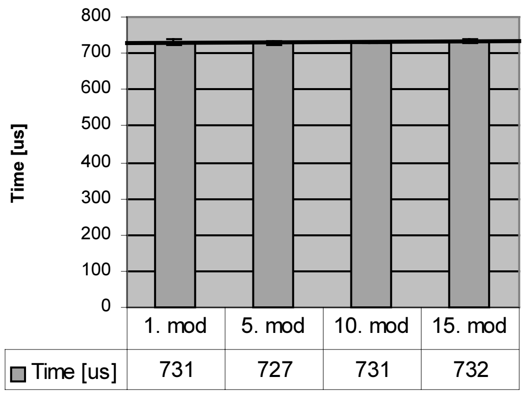
IDL.DEEP

This benchmark indicates there is no relationship between the interface nesting depth and the time it takes to complete an operation of that interface.



	1. mod	5. mod	10. mod	15. mod
▨ Time [us]	557	560	557	558

Fig. 8: *IDL.DEEP* NT AMD200 local (1 call)



	1. mod	5. mod	10. mod	15. mod
▨ Time [us]	731	727	731	732

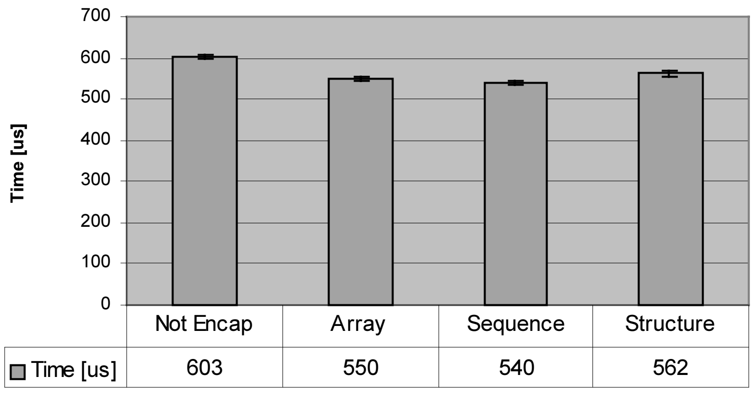
Fig. 9: *IDL.DEEP* NT Int166 -> AMD200 idle net (1 call)

6.1.1.2.3 Overall Performance

IDL.ENCAP

The IDL.ENCAP benchmark compares the invocation time when passing 100 longs individually, in an array, in a bounded sequence, and in a structure. Overall, the test shows that:

- The deviation in invocation times is within 10%, with sequence having the fastest and individual longs the slowest invocation time.



	Not Encap	Array	Sequence	Structure
▨ Time [us]	603	550	540	562

Fig. 10: *IDL.ENCAP* NT AMD200 local (1 call)

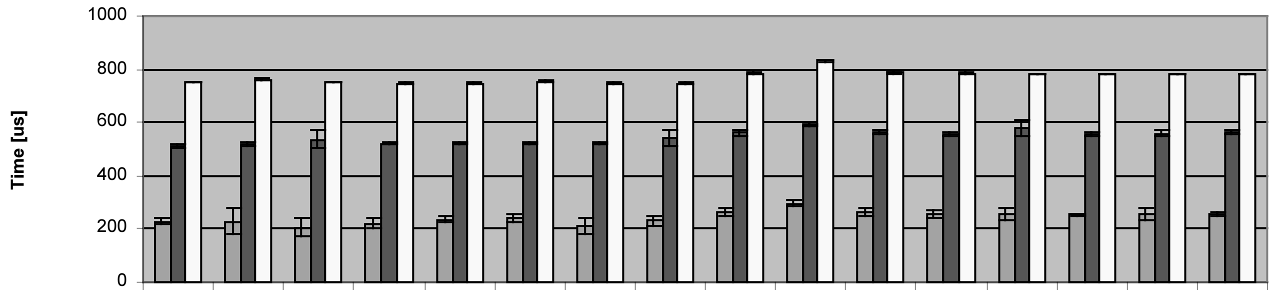
THROUGH.IN

The tests show that:

- Time to pass a basic IDL data type stays roughly the same regardless of the type being passed (deviation is about 10%)
- Time to pass an array of a basic IDL data type depends mostly on the number of items. This dependence is not exhibited for sequences.
- Encapsulating an argument within any adds a substantial overhead. Hence, when passing an array or a sequence of anys, the invocation time primarily depends on the number of anys.

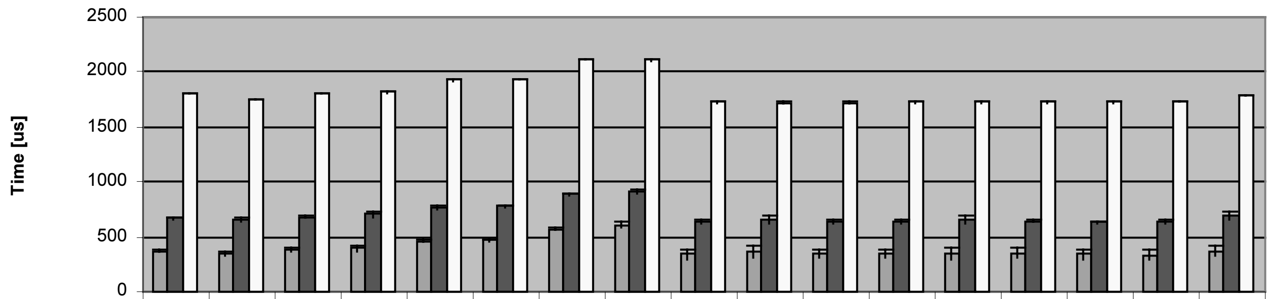
Running the same test suite over an idle 10BASE-T Ethernet shows that:

- Unless a very large number of complex arguments is passed, the constant overhead of an invocation overshadows the impact of argument sizes and types. Hence, it pays off to call less often with more arguments than vice versa.
- Local roundtrip times are roughly 206% of local delivery times (except for any data types). Remote RTT is roughly 142% of local RTT for basic data types, 254% of local RTT for arrays, 268% of local RTT for sequences. Large (1000 elements) arrays and sequences of any are passed faster over the network (upto 20% in our configuration).
- Paradoxically, local invocation times may sometimes be longer than network invocation times (e.g. when passing arrays of anys). This is caused by the high computational intensity of the marshalling and unmarshalling operations.



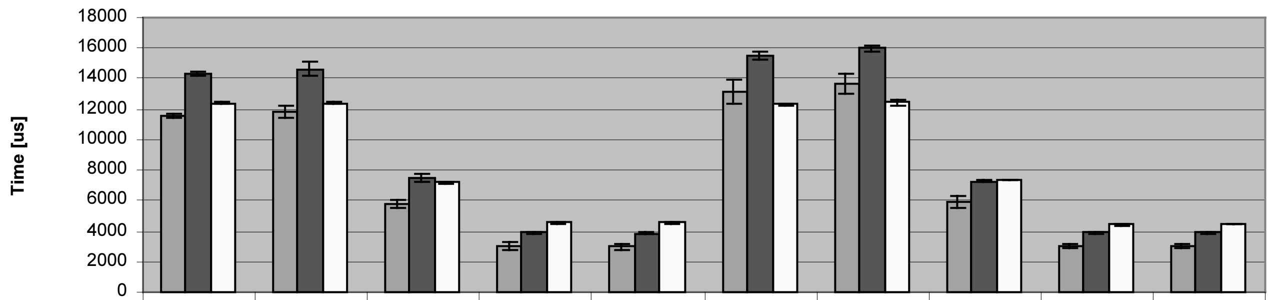
	Float	Double	Long	ULong	Short	UShort	Char	Octet	Any - Float	Any - Double	Any - Long	Any - ULong	Any - Short	Any - UShort	Any - Char	Any - Octet
▨ Local deliv.	228	226	206	220	236	240	212	230	264	296	266	258	258	252	256	254
■ Local RTT	514	518	536	520	520	520	520	540	562	590	564	558	576	558	560	562
□ Remote RTT	752	763	752	748	748	755	748	748	785	832	786	785	782	782	782	782

Fig. 11: *THROUGH.IN* NT AMD200 local and Int166 -> AMD200 idle net (time to pass **basic** data types—1 call)



	Float [256]	Double [128]	Long [256]	Ulong [256]	Short [512]	Ushort [512]	Char [1024]	Octet [1024]	seq< Float, 256>	seq< Double , 128>	seq< Long, 256>	seq< ULong , 256>	seq< Short, 512>	seq< UShort , 512>	seq< Char, 1024>	seq< Octet, 1024>	String <1024 >
▨ Local deliv.	372	354	378	396	460	470	574	608	340	366	348	350	346	350	338	334	372
■ Local RTT	668	658	680	704	766	776	888	918	640	660	640	640	656	644	636	638	686
□ Remote RTT	1806	1749	1810	1816	1928	1939	2115	2109	1726	1723	1722	1726	1728	1728	1725	1731	1793

Fig. 12: *THROUGH.IN* NT AMD200 local and Int166 -> AMD200 (idle net, time to pass **compound** data types+**string**—1 call)



	Any [1024] - Char	Any [1024] - Octet	Any [512] - Short	Any [256] - Long	Any [256] - Float	seq <Any, 1024> - Char	seq <Any, 1024> - Octet	seq <Any, 512> - Short	seq <Any, 256> - Long	seq <Any, 256> - Float
▨ Local deliv.	11548	11834	5814	3018	2996	13176	13630	5884	3054	3042
■ Local RTT	14340	14648	7474	3886	3844	15546	15990	7254	3886	3878
□ Remote RTT	12366	12406	7166	4560	4562	12304	12440	7332	4424	4423

Fig. 13: *THROUGH.IN* NT AMD200 local and Int166 -> AMD200 (idle net, time to pass **any** types—1 call)

6.1.1.3 Dynamic Type Manipulation

OmniORB provides neither the Dynamic Invocation Interface nor the Dynamic Skeleton Interface.

6.1.1.4 Repositories

OmniORB provides neither the Interface Repository nor the Implementation Repository.

6.1.1.5 Interoperability

OmniORB uses IIOP as its native communication protocol. The interoperability tests did not indicate any problem with the IIOP implementation[1]. There is no support for interoperability with Microsoft COM. Java interoperability is implied by using IIOP.

6.1.2 Nonstandard Extensions

6.1.2.1 Communication Extensions

6.1.2.1.1 Locating Objects

OmniORB provides no specialized mechanism to locate object implementations. It is, however, equipped with a fully functional Naming Service implementation that can (together with the `list_initial_services()` call) be used to locate object implementations.

6.1.2.1.2 Binding To Objects

OmniORB provides no proprietary calls for binding, the standard `string_to_object()` and `resolve_initial_references()` calls are supported though.

6.1.2.1.3 Instantiating Implementations

When creating a new object implementation to be accessible from outside of the server address space, the following steps are necessary:

(1) create a new object (`obj = new XXX_i`)

(2) register the object with `BOA->obj_is_ready(obj)` or with the `_obj_is_ready()` static member function (`obj->_obj_is_ready(BOA)`).

(3) at this point, the object reference can be exported

When deleting the object implementation, it is necessary to delete the object using the `dispose()` method, the method calls the `delete` operator upon the object that is being deleted when it is safe to do so (`BOA->dispose(obj)`)

The `BOA::obj_is_ready()` call (normally used to tell BOA the server is ready to handle requests in the unshared activation mode) has a non-standard semantics in omniORB. It should be called on all CORBA objects to notify the ORB of their existence.

6.1.2.1.4 Persistent References

When creating an object reference, it is possible to specify the object key from which the IOR is constructed. Hence, it is possible to re-create an object with the same IOR on subsequent server runs.

Using an undocumented feature of omniORB 2.5 (referred to as loaders), it is also possible to instantiate objects on demand. More information can be found at http://www.orl.co.uk/omniORB/archives/1998-Mar/0102.html.

---

[1] For a comparison table listing the results of the interoperability test, see section 9.1.1.2.

6.1.2.1.5 Automatic Activation

OmniORB has no support for an automatic activation of servers on demand.

6.1.2.1.6 Automatic Reconnection

OmniORB has no support for an automatic reconnection of a client in case of a communication failure. Whenever a failure occurs, an exception is raised.

6.1.2.1.7 Invocation Extensions

The client side proxy object can be customized in omniORB. This allows the client to change the default invocation semantics and introduce optimizations such as caching when needed. When writing the customized proxy code, the communication mechanism of the default proxy object is usually reused.

Callbacks can easily be implemented using the omniORB multi-threading support.

6.1.2.1.8 Customizability

OmniORB has no direct support for customization through hooks. As the complete source code of the ORB is freely available, an experienced programmer should be able to introduce custom modifications when necessary though.

6.1.2.2 Multi-threading Extensions

OmniORB is provided in a multi-threaded version only. The libraries shipped with omniORB are thread-safe, and are built upon the OMNIthread abstraction. The OMNIthread package is a wrapper around the underlying POSIX or POSIX-like thread package provided by the operating system.

More information about the omniORB threading model can be found at http://www.orl.co.uk/omniORB/archives/1997-Jun/0048.html.

6.1.2.2.1 Multi-threaded Servers

Although a server application can be multi-threaded (underlying libraries are thread-safe), there is no support for employing different threading strategies (see section 4.1.2.2.1) to handle incoming requests. Threads are created automatically by omniORB for every new connection (the thread-per-connection strategy).

6.1.2.2.2 Multi-threaded Clients

The omniORB client can be multi-threaded (ORB itself is thread-safe). The omniORB client locks a network connection during the remote invocation. If another thread attempts to use a locked connection for remote invocation, a new connection will be created and used for delivering the request.

MT.CLIENT

The MT.CLIENT benchmark take 50 seconds to complete, indicating that omniORB clients can issue parallel requests (and receive replies) over multiple connections.

MT.CONN

The MT.CONN benchmark shows that omniORB is capable of parallelizing requests from multiple threads (the benchmark takes 50 seconds to complete). The threads that run in parallel open additional network connection—on one hand, this slows down the performance a bit, on the other hand, the client is able to issue requests in parallel even to servers that use the "thread per connection" strategy (e.g. omniORB servers).

This benchmark indicates a problem with the multi-threading support at the client side of the application. When more than five threads issue a request to the same server in parallel, the client crashes (there is no such limitation when communicating with more servers).

6.1.2.2.3 Multi-threaded ORB

A network connection is created on demand (on the client side) and closed automatically when it is idle for a given period of time (on the server side). Inside the ORB, two separate threads are dedicated to scanning for idle connections (one thread is responsible for outgoing connections and the other looks after incoming connections). The thread for incoming connections is only created when BOA is initialized, because only then an incoming connection may occur. The inactivity period of the two threads can be configured by the `idleConnectionScanPeriod()` call.

6.1.2.2.4 Concurrency

Both at the client and at the server side, it is the application programmer who is responsible for protecting shared data structures at the application level (using the underlying operating system's synchronization primitives, or those from OMNIthread package).

6.1.2.3 Management Extensions

OmniORB does not provide any management utilities. The configuration of the ORB itself is very simple, as it does not support dynamic activation policies (the server application must be running before a client request arrives at the server host).

6.1.3 Scalability

6.1.3.1 Speed vs. Number of Objects

PROXY

This benchmark indicates dependence between the number of objects and the object creation time both on the server and the client sides. There is also a significant difference between the time it takes to invoke an operation for the first time and for all subsequent times.

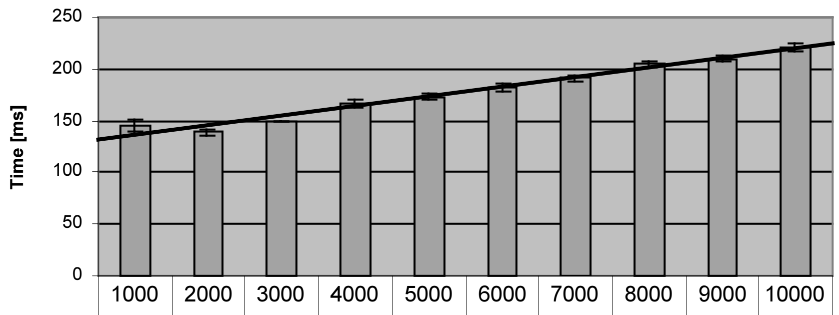
PROXY	**NT local**			**NT idle net**		
objects	1000	2000	3000	1000	2000	3000
CREATE	105	204	311	121	233	350
FIRST	959	1950	3035	1381	2724	4187
NEXT	464	1001	1566	717	1414	2159

Fig. 14: Table *PROXY* NT AMD200 and Int166 -> AMD200 idle net (time in ms)

PROXY.LOT

The PROXY.LOT benchmark shows that:

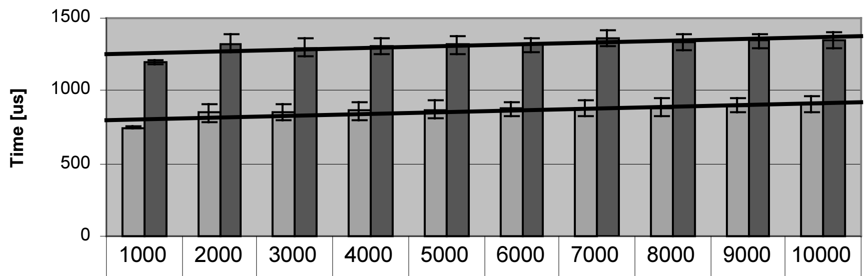
- There is a linear dependence between the number of objects handled by the server and the time it takes to create and register a new object. The dependence becomes significant with thousands of objects handled.
- There is no measurable dependence between the number of objects handled by the client and the time it takes to receive a new object reference.
- There is a linear dependence between the number of objects handled by the server and the time it takes to invoke a request on the server. The dependence becomes significant with thousands of objects handled.



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ Get	146	139	149	167	173	182	191	205	210	221

Objects on server & proxies on client

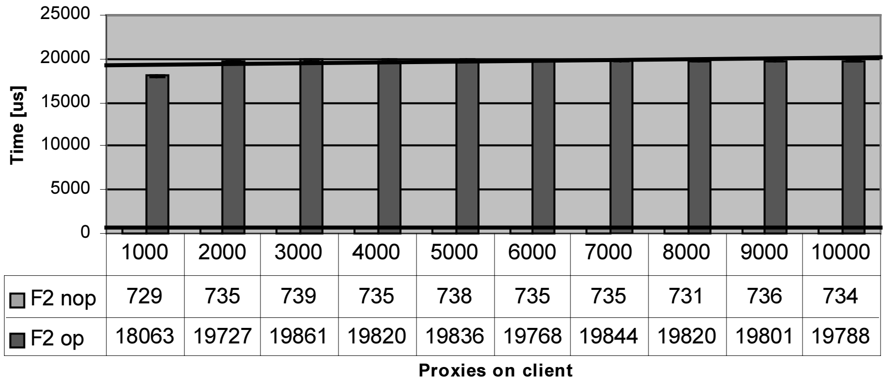
Fig. 15: *PROXY.LOT* NT Int166 local



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ F1 nop	747	849	854	863	873	874	883	886	901	906
■ F1 op	1193	1325	1300	1310	1317	1315	1365	1337	1344	1347

Objects on server & proxies on client

Fig. 16: *PROXY.LOT* NT Int166 local



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ F2 nop	729	735	739	735	738	735	735	731	736	734
■ F2 op	18063	19727	19861	19820	19836	19768	19844	19820	19801	19788

Fig. 17: *PROXY.LOT* NT Int166 local

6.1.3.2 Resources vs. Number of Objects

For 10000 objects, the client consumes 313 bytes per object proxy, the server consumes 238 bytes per object (the most trivial implementation object with no attributes and one operation `void x()`).

6.1.3.3 Resources vs. Queued Requests

ONEWAY

When testing the number of oneway void calls that can be issued to an omniORB server, the results indicated that regardless of the client used (omniORB, Orbix and VisiBroker clients were tested), about 500 oneway calls with no arguments can be issued before the client blocks.

6.1.4 Robustness

No unexpected crashes and no bugs were encountered in omniORB under normal usage conditions. The ORB exhibited a stable performance in our tests, except for occasional server crashes on AIX (these might have been caused by an incorrect AIX port though).

6.1.4.1 Reliable Servers

OmniORB provides no means for automatic invocation of servers on demand. Although in principle possible, it would be difficult for the application programmer to write a useable substitute for the automatic invocation mechanism.

To provide restartable servers, a programmer can use persistent object references. A server can assign the same object reference to its objects every time it starts, a support for object activation on demand is provided as well.

The omniORB LifeCycle support can be used to move object implementations, e.g. for the purposes of load balancing. The forwarding mechanism, however, relies on the original object server being able to redirect all incoming requests to the new object server (and this for a potentially infinite period of time). Thus, it would be difficult to use this mechanism to improve reliability by replication.

The Naming Service can be used to find multiple implementations of the same object, this can be used in load balancing. The limited LifeCycle support can be used for moving of object implementations.

6.1.4.2 Limits

SIZE

By default, omniORB limits the maximum size of the message that can be sent or received, to 2MB. The limit can be queried and set using the `omniORB::MaxMessageSize()` call. If an incoming message exceeds this limit, it is not unmarshaled and the COMM_FAILURE exception is raised. If an outgoing message exceeds this limit, the MARSHAL exception is raised.

Number of Objects

The OmniORB 2.5.0 server works OK with 250000 objects (more objects were not tested), the client works OK with 50000 objects (more objects were not tested).

6.1.4.3 Bugs

6.1.4.3.1 Long IDL names bug

OmniORB 2.5 has problems with IDL names longer than 500 characters (IDL compiler does not compile the IDL source).

6.1.4.3.2 Long IDL interfaces bug

OmniORB 2.5 has problems with IDL interfaces that contain a large number of operations (the C++ sources generated by the IDL compiler do not compile under VC++ 4.2 if more than 95 operations are declared within a single interface, the limit is slightly higher for VC++ 5.0).

6.1.4.3.3 Parallel connections bug

The MT.CONN benchmark indicates a problem with the multi-threading support at the client side of the application. When more than five threads issue a request to the same server in parallel, the client crashes (there is no such limitation when communicating with more servers).

6.1.4.3.4 Stability problems

OmniORB 2.5 on AIX occasionally crashed in the THROUGH.IN and PROXY.LOT benchmarks. The NT version, however, has been stable throughout all tests. It is worth noting that the AIX port is not officially supported by ORL.

6.2 Object Services

Of the standardized services, only the Naming Service (omniNames) is provided. A partial support for the Lifecycle Service is also incorporated within omniORB.

6.2.1 Lifecycle Service

As a part of omniORB, a support for moving and deleting of objects is provided. The lifecycle support automates creation of proxies capable of redirecting requests to objects that have been moved, making it possible to transparently (with respect to clients) migrate objects on request.

6.2.2 Naming Service

The implementation of the Naming Service in omniORB is called omniNames. OmniNames claims full compliance with the OMG Naming Service specification and provides all interfaces defined by the standard except for the names library (which is not vital to the service functionality).

The OmniNames server is persistent—both the database of name-to-object bindings and the object reference of the server are preserved across server invocations. This allows the OmniNames server to be restarted transparently in case of failure, thus increasing the overall robustness of the system.

The `rebind()` call is implemented incorrectly. It fails (`NotFound` exception is raised) when the caller is not bound to the Naming Service. To conform to the Naming Service specification, the service should simply `bind()` to the Naming Service instead of raising an exception. The workaround is simple (catch the exception and try to call `bind()`).

6.2.3 Persistence Service

There is no implementation of a separate Persistence Service for omniORB. All by itself, omniORB supports persistent object references, but does not support launching servers on demand. Although this allows persistent servers to be implemented, the process incurs significant amount of coding. An undocumented feature of the omniORB is the concept of loaders (mechanism for activating object implementations when a request arrives)—see the section 6.1.2.1.4.

6.3 Miscellanea

6.3.1 Requirements

Currently, ORL provides omniORB for the following operating systems and compilers:

- Solaris 2.5/ Sun SparcCompiler C++ 4.2
- Digital Unix 3.2/ DEC C++ compiler 5.5
- Linux 2.0 (x86)/ GNU C++ compiler 2.7.2 / Linuxthreads 0.5
- MS Windows NT / MS Windows 95 / MS Visual C++ 5.0

OmniORB for these platforms is provided in a binary form (sources are available as well). Ports to following platforms are done by external contributors and are not provided in binary form:

- IBM AIX 4.2/ IBM C Set++ 3.1.4
- HPUX 10.20/ aC++ (B3910 A.01.04)
- OpenVMS Alpha 6.2/ DEC C++ compiler 5.5 (UCX 4.1 ECO 8)
- OpenVMS Vax 6.1/ DEC C++ compiler 5.5 (UCX 4.0 ECO 1)
- NextStep 3.3/ gcc-2.7.2

It should be straightforward to port omniORB to any platform that supports POSIX-style threads, BSD-style sockets and has a C++ compiler that supports exceptions.

6.3.2 Development

The quality of documentation is intermediate (perhaps most missing is the reference guide). The User's Guide, a description of the OMNIthread abstraction, a description of the OMNI Naming Service and LifeCycle support, and a short description of the omniORB utilities are provided. In spite of brevity, the documentation is sufficient. When all else fails, the ORB sources are available.

Useful features with respect to debugging include activity logging, range checking on sequence types, safety checks on reference counts, and object reference parsing utilities.

Technical support is not provided, omniORB is free. Queries, suggestions, or problems concerning omniORB can be sent to ORL at `omniorb@orl.co.uk`. ORL maintains a mailing list dedicated to omniORB. More information about the list is at http://www.orl.co.uk/omniORB/mailingList.html.

6.3.2.1 Development Support

Feature		
Access to the reference count value	No	reference count is private attribute
Warn when negative reference count	**Yes**	
Range checking on sequences	**Yes**	exception raised
Generation of skeleton implementation	No	
Symbolic exception dumping to stream	No	
Broker activity logging	**Yes**	very verbose logging possible
Reference parsing utilities	**Yes**	`genIOR` and `catIOR`

6.3.3 Vendor

6.3.3.1 Pricing

OmniORB is copyright Olivetti & Oracle Research Laboratory. It is a free software provided under the GNU General Public License version 2 (or later) as published by the Free Software Foundation.

OmniORB contains source code that is derived from the SunSoft OMG IDL Compiler Front End (CFE) version 1.3. The IDL CFE is copyright Sun Microsystems and is distributed under conditions similar to those of GNU.

6.3.3.2 Company

Established in 1986 as Olivetti Research Laboratory, ORL is now a jointly funded research laboratory of Ing. C. Olivetti & C., S.p.A, and the Oracle Corporation. Over 50 research staff are employed at the company's facilities in central Cambridge, England. The company specializes in networking and communications, multimedia, mobility, and distributed systems.

7 Iona Orbix

Broker:	**Orbix**
Vendor:	**Iona Technologies PLC**
Version tested:	**Windows NT:** ▪ **Orbix 2.2c01 Desktop** ▪ **Orbix 2.3c Desktop** **AIX:** ▪ **Orbix 2.2MT AIX**
Platforms used:	**Windows NT 4.0 Workstation** ▪ **200MHz AMD K5 64MB RAM** ▪ **166MHz Intel Pentium 64MB RAM** ▪ **179MHz Intel PentiumPro 64MB RAM** **AIX 4.1 on IBM RS6000-595** **network: 10BASE-T Ethernet** **compilers:** ▪ **Microsoft Visual C++ 4.2** ▪ **IBM C Set ++ 3**

As one of the first implementations of the CORBA standard, Orbix is a widely used ORB produced by IONA Technologies PLC. The chief characteristics of Orbix are a wide range of supported platforms and availability of a number of accompanying products (e.g. object services).

7.1 Object Request Broker

7.1.1 Standardized Functionality

7.1.1.1 Interface Definition Language

Orbix IDL conforms to the CORBA 2.0 specification. The set of supported data types is extended by allowing opaque data types to be defined using a new IDL keyword `opaque`. An argument of an opaque data type is always passed by value, the value itself is marshaled and unmarshaled by user-supplied code.

Orbix is a CORBA implementation for C++. Other IONA products (not tested here) support mappings from IDL to Java, Ada, C, COBOL, and Smalltalk. The IDL-to-C++ mapping conforms to the CORBA 2.0 specification.

7.1.1.2 Basic Remote Invocation

Orbix is a full implementation of the CORBA 2.0 standard. Orbix provides complete ORB, BOA, DII, DSI, Interface Repository (IR) and Implementation Repository modules. Orbix supports both the “at most once, synchronous” and the “best effort, asynchronous” invocations defined by the standard. It also supports server activation policies, exceptions and context handling.

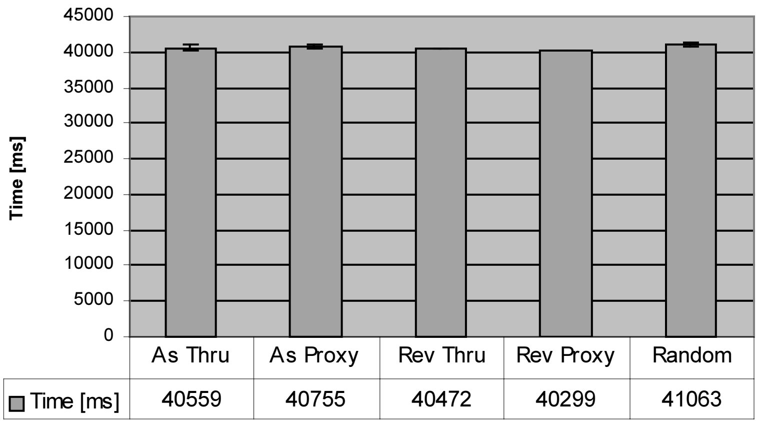
The Orbix implementation of both the ORB and the BOA interfaces conforms to the standard. The semantics of the BOA `dispose()` call differs from the standard—it unregisters the object implementation with BOA (as specified by the standard) and deletes the object. Apart from using `dispose()`, an object implementation can also be deleted by decreasing its reference count to zero using `release()`.

Orbix supports IDL exceptions[1] regardless of whether the C++ compiler offers native exception handling or not. For compilers that do not support exceptions, a set of macros is provided as a replacement for the try-catch clauses; the information about exceptions is passed within an environment as a part of each call. Orbix supports context handling.

7.1.1.2.1 Measurements Verification

CALL

The results of the CALL test indicate there is no significant relationship between the pattern of the calls issued to the server and the times measured by the benchmark. This means that it is possible to apply the results of other benchmarks to applications with different invocation patterns.



	As Thru	As Proxy	Rev Thru	Rev Proxy	Random
▨ Time [ms]	40559	40755	40472	40299	41063

Fig. 18: *CALL* NT Int179 (10.000 calls)

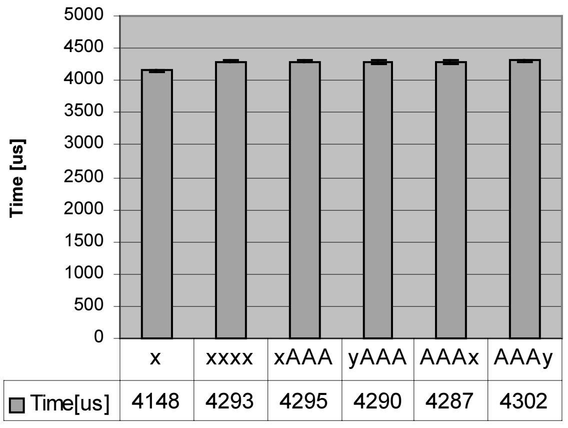
7.1.1.2.2 Dispatcher Performance

IDL.NAMES

The most important result from the IDL.NAMES benchmark is the demonstration of the fact that calling operations with longer (tested for 500 characters) names takes longer time to complete. This is due to the fact that the marshalling, unmarshalling, transfer and dispatching algorithms all use a plain string form of the operation names. The test shows that:

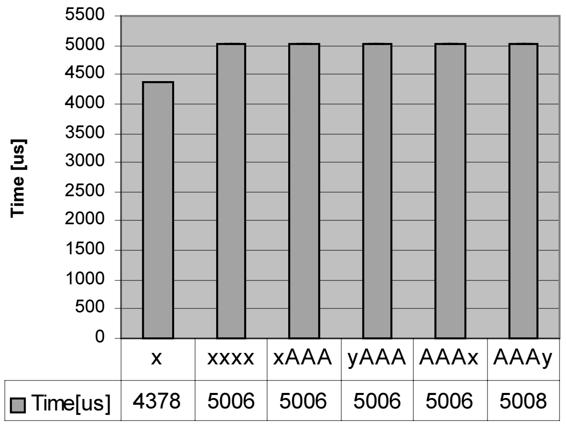
- For normal-length names (10-30 characters), the difference in invocation time is insignificant (about 1%).
- The difference in invocation time is more significant when running over a network (more data is passed).

[1] Note that user defined exceptions are not supported when using DII.



	x	xxxx	xAAA	yAAA	AAAx	AAAy
▨ Time[us]	4148	4293	4295	4290	4287	4302

Fig. 19: *IDL.NAMES* NT AMD200 local (1 call)



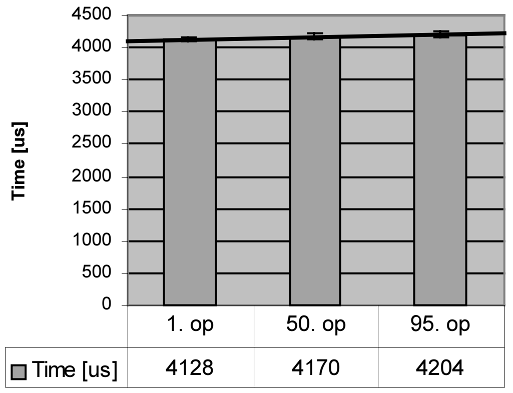
	x	xxxx	xAAA	yAAA	AAAx	AAAy
▨ Time[us]	4378	5006	5006	5006	5006	5008

Fig. 20: *IDL.NAMES* NT Int166 -> AMD200idle net (1 call)

IDL.LONG

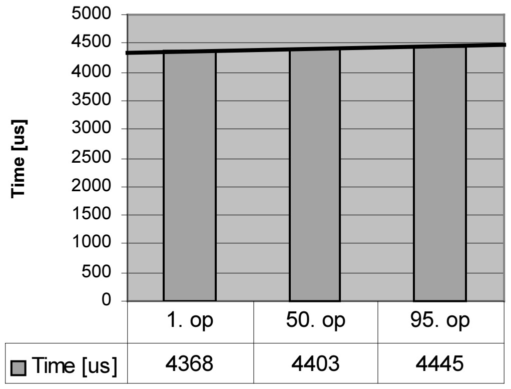
The IDL.LONG benchmark shows that the operation that is defined later in the IDL interface takes longer time to complete. The reason for this is that the dispatcher code (generated from the IDL definition) looks up the operation name using a sequential (linear) search. The test shows that:

- For medium-sized interfaces (100 operations), the difference in invocation time between the first and the last operation may be significant.
- The difference in invocation time does not change when running over a network (clearly, only server side code matters).



	1. op	50. op	95. op
▨ Time [us]	4128	4170	4204

Fig. 21: *IDL.LONG* NT AMD200 local (1 call)



	1. op	50. op	95. op
▨ Time [us]	4368	4403	4445

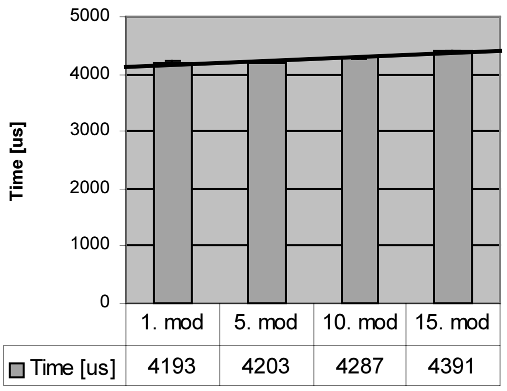
Fig. 22: *IDL.LONG* NT Int166 -> AMD200 idle net (1 call)

IDL.DEEP

The IDL.DEEP benchmark indicates a relationship between the interface nesting depth and the time it takes to complete an operation of that interface. The test shows that:

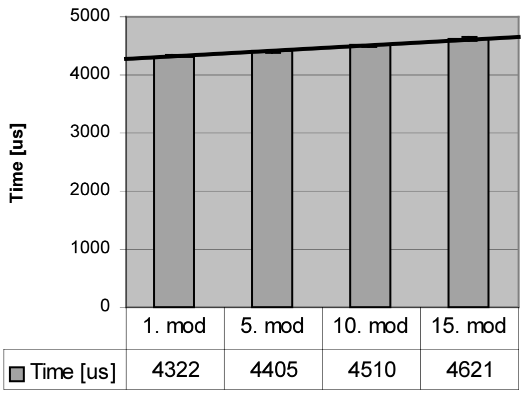
- The difference in invocation time between operations in non-nested and nested interfaces is significant only for relatively deeply nested interfaces (10 levels).

- The difference in invocation time is more significant when running over a network (more data is passed).



	1. mod	5. mod	10. mod	15. mod
▨ Time [us]	4193	4203	4287	4391

Fig. 23: *IDL.DEEP* NT AMD200 local (1 call)



	1. mod	5. mod	10. mod	15. mod
▨ Time [us]	4322	4405	4510	4621

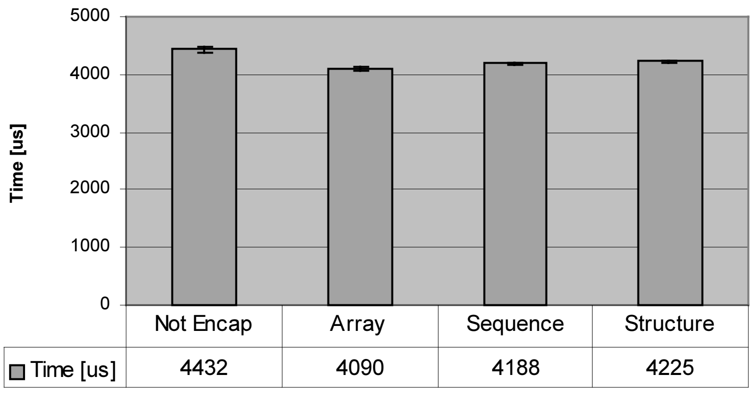
Fig. 24: *IDL.DEEP* NT Int166 -> AMD200 idle net (1 call)

7.1.1.2.3 Overall Performance

IDL.ENCAP

The IDL.ENCAP benchmark compares the invocation time when passing 100 longs individually, in an array, in a bounded sequence, and in a structure. Overall, the tests show that:

- The deviation in invocation times is usually within 10%, with arrays having the fastest and individual longs the slowest invocation time.



	Not Encap	Array	Sequence	Structure
▨ Time [us]	4432	4090	4188	4225

Fig. 25: *IDL.ENCAP* NT AMD200 local (1 call)

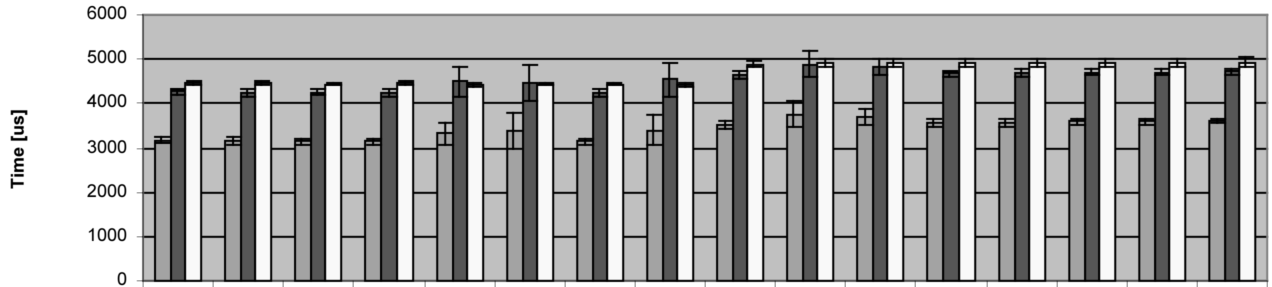
THROUGH.IN

The tests show that:

- Time to pass a basic IDL data type stays roughly the same regardless of the type being passed.
- Time to pass an array or a sequence of a basic IDL data type depends mostly on the length of data in octets.
- Encapsulating an argument within any adds a substantial overhead. Hence, when passing an array or a sequence of anys, the invocation time primarily depends on the number of anys.

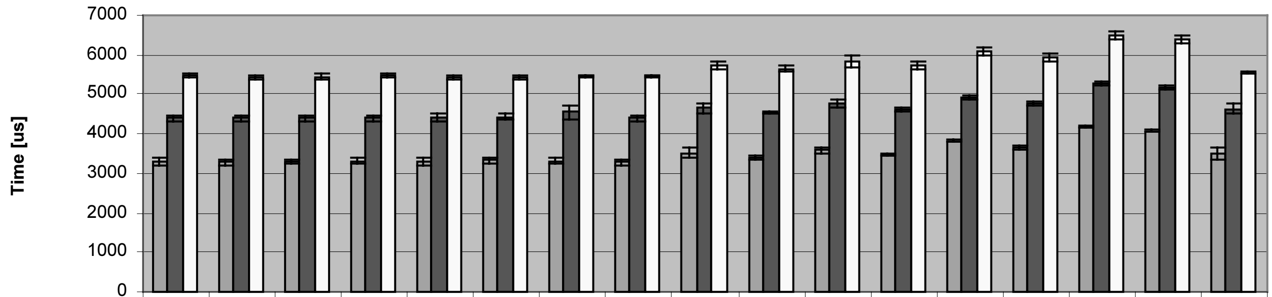
Running the same test suite over an idle 10BASE-T Ethernet shows that:

- Unless a very large number of complex arguments is passed, the constant overhead of an invocation overshadows the impact of argument sizes and types. Hence, it pays off to call less often with more arguments than vice versa.
- Local roundtrip times are roughly 133% of local delivery times (except for any data types). Remote RTT is roughly 103% of local RTT for basic data types, 123% of local RTT for arrays and sequences. Large (1000 elements) arrays and sequences of any are passed faster over the network (about 16% in our configuration).
- Paradoxically, local invocation times may sometimes be longer than network invocation times (e.g. when passing arrays of anys). This is caused by computational intensity of marshalling and has been explained by IONA in the Knowledge Base (article number 434.994).



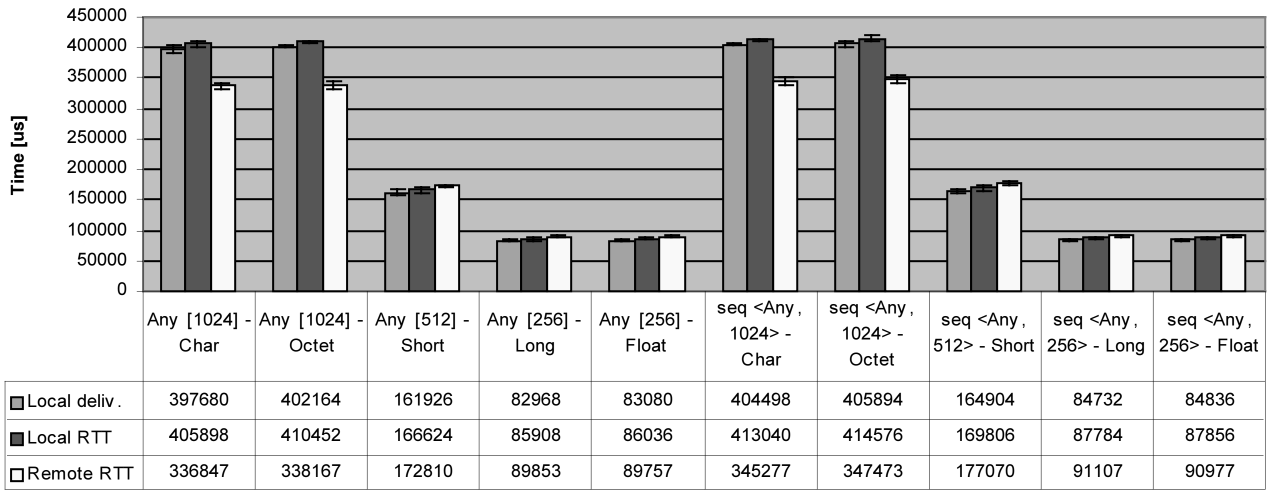
	Float	Double	Long	ULong	Short	UShort	Char	Octet	Any - Float	Any - Double	Any - Long	Any - ULong	Any - Short	Any - UShort	Any - Char	Any - Octet
▨ Local deliv.	3178	3166	3140	3146	3324	3370	3148	3384	3532	3760	3692	3554	3562	3598	3592	3608
■ Local RTT	4264	4258	4252	4252	4494	4466	4248	4534	4660	4890	4824	4674	4682	4712	4704	4722
□ Remote RTT	4453	4457	4443	4447	4440	4443	4437	4440	4887	4937	4917	4907	4923	4930	4927	4937

Fig. 26: *THROUGH.IN* NT AMD 200 and Int166 -> AMD200 idle net (time to pass **basic** data types—1 call)



	Float [256]	Double [128]	Long [256]	Ulong [256]	Short [512]	Ushort [512]	Char [1024]	Octet [1024]	seq< Float, 256>	seq< Double , 128>	seq< Long, 256>	seq< ULong , 256>	seq< Short, 512>	seq< UShort , 512>	seq< Char, 1024>	seq< Octet, 1024>	String <1024 >
▨ Local deliv.	3296	3280	3284	3306	3284	3326	3314	3280	3520	3416	3578	3474	3810	3668	4180	4080	3492
■ Local RTT	4396	4402	4400	4406	4410	4428	4546	4398	4642	4538	4750	4612	4932	4776	5274	5188	4632
□ Remote RTT	5457	5443	5443	5467	5443	5440	5457	5453	5733	5653	5850	5747	6070	5937	6500	6393	5547

Fig. 27: *THROUGH.IN* NT AMD 200 and Int166 -> AMD200 idle net (time to pass **compound** data types+**string**—1 call)



	Any [1024] - Char	Any [1024] - Octet	Any [512] - Short	Any [256] - Long	Any [256] - Float	seq <Any, 1024> - Char	seq <Any, 1024> - Octet	seq <Any, 512> - Short	seq <Any, 256> - Long	seq <Any, 256> - Float
▨ Local deliv.	397680	402164	161926	82968	83080	404498	405894	164904	84732	84836
■ Local RTT	405898	410452	166624	85908	86036	413040	414576	169806	87784	87856
□ Remote RTT	336847	338167	172810	89853	89757	345277	347473	177070	91107	90977

Fig. 28: *THROUGH.IN* NT AMD 200 and Int166 -> AMD200 idle net (time to pass **any** types—1 call)

7.1.1.3 Dynamic Type Manipulation

Orbix fully supports DII and DSI. In addition to the standard DII and DSI calls, it also provides a stream-based user-friendly interface to both DII and DSI. User defined exceptions are not supported when using DII.

7.1.1.4 Repositories

The Interface Repository is provided as a standalone server, capable of parsing IDL definitions when launched or on-the-fly when queried. The repository complies with the CORBA specification.

All standard server activation policies are supported by the Implementation Repository. The Implementation Repository also provides a level of security by protecting server launch and invoke operations by access rights based on user names and hosts.

7.1.1.5 Interoperability

Starting from version 2.3, Orbix uses the IIOP protocol as default (earlier versions use a proprietary protocol, called POOP—Plain Old Orbix Protocol, but all versions starting from 2.0 support IIOP).

The IIOP protocol implementation inside the Orbix daemon has problems responding to the IIOP 1.0 Location Request message, thus the Orbix 2.2 and 2.3 servers on Windows NT appear to be unable to interoperate with other than Orbix clients (this can be worked around but the solution is cumbersome[1] and disables the automatic server activation mechanism). Orbix 2.2 on AIX has problems passing sequences. The detailed results of the interoperability test can be found in section 9.1.1.2.

[1] The workaround that worked in our tests was based on substituting the real server port for the demon port in the object references. According to IONA technical support, this can be done automatically using the undocumented `useTransientPort()` call, but in our tests this call caused the server to crash. The latest patch for Orbix 2.3 (#12) should fix the problem as well.

Orbix for Windows has a built-in support for interoperability with OLE/COM/ActiveX. Orbix programmer can implement any combination of CORBA/COM server/client. Orbix is able to communicate with Java applications using the IIOP protocol. IONA also provides a Java-based ORB called OrbixWeb.

7.1.2 Nonstandard Extensions

7.1.2.1 Communication Extensions

7.1.2.1.1 Locating Objects

To locate object implementations, Orbix provides a proprietary solution called Locator. When using the Locator, a client asks for available object implementations by giving a service name. The Locator then returns a list of hosts running the given service.

Orbix comes with a default Locator implementation that uses a static configuration file (the file contains tuples of service names and addresses of hosts that can potentially run the service). A host that does not run the advertised service can forward incoming Locator requests according to their configuration file. A client programmer can replace the default Locator implementation.

7.1.2.1.2 Binding To Objects

The `_bind()` call in Orbix allows the client side code to bind to an object of a given interface even when the server or the object key is not specified. The actual binding can be deferred until the first request is sent (`CORBA::ORB::pingDuringBind()`).

7.1.2.1.3 Instantiating Implementations

On the Orbix server, all instances of the `CORBA::Object` class are registered automatically by the ORB at the time of their creation. Hence, it is not necessary to register objects explicitly (indeed, no call for explicit object registration is provided).

To delete an object implementation, the server programmer should decrease its reference count to zero. When the count reaches zero, Orbix releases the resources associated with the object implementation and deletes the implementation object. It is also possible to call the `BOA::dispose()` operation to delete the object implementation (but this is not recommended in the documentation). By default, it is not possible to delete an object implementation using the C++ `delete` operator (an exception is raised), but this behavior can be changed using the `set_unsafeDelete()` call.

7.1.2.1.4 Persistent References

In Orbix, persistent object references are provided by a mechanism based on Loaders. A Loader is consulted whenever an inactive object implementation is contacted by a client, and whenever the object implementation is about to be deleted. By supplying its own loader, the server programmer can specify the behavior of the ORB in these two situations (e.g. create the object implementation to be contacted, or save the current state of the object implementation to be deleted).

7.1.2.1.5 Automatic Activation

The Orbix daemon is able to launch a server automatically whenever a request for the server comes in and the server is not running. Orbix 2.2c01 automatic activation only works on the servers that use the proprietary Orbix protocol. This is most likely a bug, and is fixed in Orbix2.3c, where the automatic activation works OK.

7.1.2.1.6 Automatic Reconnection

If the connection is lost while the server is idle, it is re-established transparently (the client will not know about the failure). If the connection is lost while the server processes a call, an exception is raised and the connection can be re-established simply by retrying the call (the operation completion status is returned as a part of the exception). This behavior is not documented though and is valid for multi-threaded versions of Orbix only, single-threaded Orbix throws the COMM_FAILURE exception whenever the connection is lost and re-establishes the connection when the call is retried.

The automatic reconnection can be turned off using the `CORBA::ORB::noReconnectOnFailure()` call. The timeout for a single connection attempt and the number of maximum retries can be set as well.

7.1.2.1.7 Invocation Extensions

The client side proxy object can be customized in Orbix by supplying a proxy factory (multiple factories can be installed). This allows the client to change the default invocation semantics and introduce optimizations such as caching when needed. The default proxy object is usually reused to provide the communication support.

Callbacks can easily be achieved using the Orbix multi-threading support. Single-threaded Orbix versions can implement callbacks as well, this is done by the client explicitly polling for events during the remote invocation and requires extra coding from the application programmer.

7.1.2.1.8 Customizability

In Orbix, various hooks along the request processing path can be set using filters and request transformers. In addition to this, the server event processing loop (`obj_is_ready()` and `impl_is_ready()`) can be integrated with a foreign events handler.

The Orbix filters can be used for filtering and adding data to requests. Filters are of two kinds: per-process (all requests within a process are filtered) and per-object (only requests for a specific object are filtered). The two kinds of filters can co-exist within a single server, in such a case the per-object filters are architecturally closer to the object implementation (are invoked after per-process filters when an invocation arrives and before per-process filters when the invocation returns). Note that the per-process filters cannot access arguments of the requests—only the operation name and the target object reference is available.

Per-process filters can be installed in eight points along the request processing path:

- *client side* - before and after marshalling data into an outgoing request, before and after unmarshalling data from an incoming reply
- *server side* - before and after unmarshalling data from an incoming request, before and after marshalling data into an outgoing reply

When the client and the server are collocated in the same address space, only per-object filters can be used. Per-object filters require the implementation to be bound to the skeleton class using the TIE approach.

Another way to modify incoming or outgoing requests (by adding data new data or by modifying the existing data within the request) is provided by request transformers. Transformers are invoked upon all sent or received requests within a process. Unlike filter, a transformer does not operate directly upon the `Request` class, but only upon a data buffer (array of octets).

Orbix supports integration of the server event processing loop with a foreign events handler. There are two ways to achieve this integration—either the foreign event handler polls for Orbix events (see the `processEvents()` set of calls), or the code that polls for foreign events is called from within the Orbix event loop (see the `addForeignFD()` set of calls). This is especially useful in single-threaded applications.

7.1.2.2 Multi-threading Extensions

Orbix is available in two basic versions: single-threaded (Orbix) and multi-threaded (OrbixMT)[1]. The only significant difference between the two versions is that all libraries provided with OrbixMT are thread-safe (an application code written for the single-threaded Orbix will execute correctly if linked with OrbixMT libraries).

The thread-safe libraries are built upon the underlying operating system's thread package. Both the client and the server can be multi-threaded (provided their code also relies on the operating system's thread package). Multi-threaded and single-threaded clients and servers can be combined in any fashion.

7.1.2.2.1 Multi-threaded Servers

An Orbix server can support all of the threading strategies listed in section 4.1.2.2.1. Using the request filtering mechanism (special per-process filter of the `CORBA::ThreadFilter` class), Orbix allows the server programmer to arbitrarily create threads to serve incoming requests or to pass the request to any of the existing threads. Thus, the programmer can choose the best-fitting strategy for dispatching of incoming requests.

The flexibility of this approach is paid for by the fact that none of the threading strategies from section 4.1.2.2.1 is supported automatically (the programmer has to implement them by hand by inheriting from the `CORBA::ThreadFilter` base class and by rewriting the `inRequestPreMarshall()` method). On the other hand, it is not difficult to implement the most common strategies (e.g. pool of threads) once and reuse the code whenever it is useful (e.g. as a library)[2].

The MT.POOL test verifies the ability of the ORB to create pool of threads. The implementation of the threading strategy is fully in the server programmer's hands.

7.1.2.2.2 Multi-threaded Clients

The Orbix client can be multi-threaded. Both the MT.CLIENT and the MT.CONN benchmarks take 50 seconds to complete, indicating that Orbix clients can issue parallel requests both over single and multiple connections.

7.1.2.2.3 Multi-threaded ORB

At the ORB level, a single TCP/IP connection is used to transfer all requests between two address spaces. The application programmer can limit the number of threads handling the incoming connections (connection threads) by using the `CORBA::ORB.maxConnectionThreads()` operation, and the number of connections per connection thread by using the `CORBA::ORB.maxFDsPerConnectionThread()` call.

7.1.2.2.4 Concurrency

Both at the client and at the server side, it is the application programmer who is responsible for protecting shared data structures at the application level (using the underlying operating system's synchronization primitives).

As for locking the ORB's internal data structures, a programmer can control the locking of every `CORBA::Object` instance using its `_enableInternalLock()` operation.

---

[1] Orbix for Windows is shipped with both the single-threaded and the multi-threaded libraries.

[2] For an example implementation, see the MT.POOL benchmark.

7.1.2.3 Management Extensions

Alone, the Orbix installation provides a set of command line utilities to manage the Implementation Repository, used by the Orbix daemon to implement dynamic activation policies (apart from the dynamic activation, the Implementation Repository also provides a level of security by requiring each server to be registered with explicit launch and invoke rights based on user and host names).

The basic installation of Orbix 2.3c Desktop also contains the GUITools 3.1 package, functionally similar to the command line utilities mentioned above encapsulated in a comfortable graphical suite. The GUITools 3.1 package contains the following tools:

- *Orbix Configuration Tool* - for basic configuration of the local ORB installation (network configuration, directories, logging and other basic settings).
- *Orbix Interface Repository Browser* - to browse the contents of the Interface Repository server (see section 5.1.1).
- *Orbix Naming Service Browser* - to browse the contents of the Naming Service database (see section 5.1.2.2).
- *Orbix Server Manager* - to manage the Implementation Repository, and also to configure remote Orbix object implementations (provided the user has the necessary rights).

A separate management package, called OrbixManager, is also available. Through the graphical user interface of the OrbixManager suite, a system administrator can monitor and manage the Orbix applications running on various hosts throughout the network. In addition to the graphical interface, OrbixManager also provides access to the monitoring and management information through the Simple Network Management Protocol (SNMP) Management Information Base (MIB). Hence, the system administrator can manage Orbix using any SNMP system management platform.

Apart from allowing the system administrator to access the configuration and management information from a single location, OrbixManager also monitors Orbix components and notifies the administrator about unexpected events, such as server shutdowns or exceptions (to notify the management service of an exception, the application must be linked with a management library). OrbixManager can also collect statistical information (e.g. number of invocations, number of exception raised, throughput, number of connections, queue lengths) from a running application.

7.1.3 Scalability

7.1.3.1 Speed vs. Number of Objects

7.1.3.1.1 PROXY

The PROXY.CREATE benchmark indicates dependence between the number of objects and the object creation time both on the server and the client sides. The PROXY.FIRST and PROXY.NEXT benchmarks show that:

- There is a significant difference between the time it takes to invoke an operation for the first time and for all subsequent times.

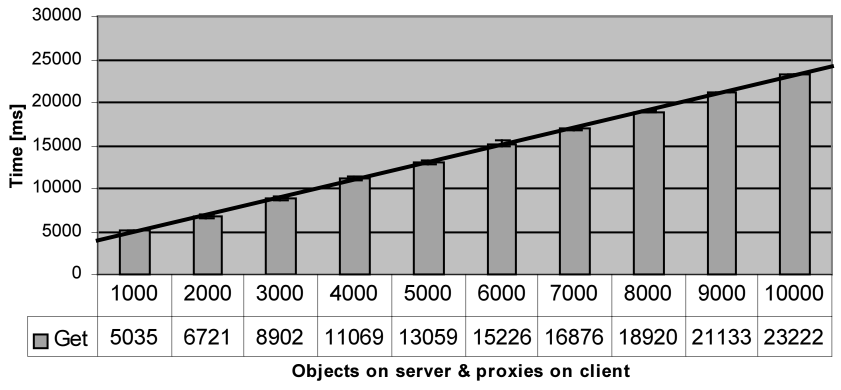
PROXY	**NT local**			**NT idle net**		
objects	1000	2000	3000	1000	2000	3000
CREATE	8949	27154	59816	12394	37036	82242
FIRST	4706	9611	14496	5167	10266	16089
NEXT	4393	8477	12804	4462	9601	13514

Fig. 29: Table *PROXY* NT AMD200 and Int166 -> AMD200 idle net (time in ms)

7.1.3.1.2 PROXY.LOT

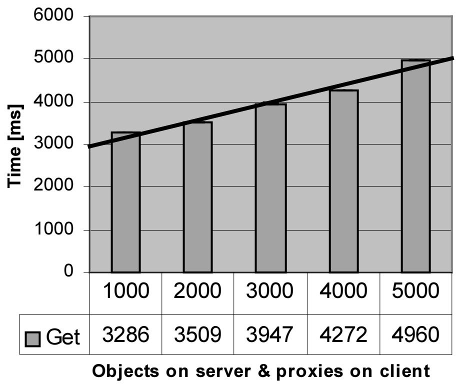
The PROXY.LOT benchmark shows that:

- There is a linear dependence between the number of objects handled by the server and the time it takes to create and register a new object. The dependence becomes significant with hundreds of objects handled.
- There is a linear dependence between the number of objects handled by the client and the time it takes to receive a new object reference. The dependence becomes significant with hundreds of objects handled.
- There is a linear dependence between the number of objects handled by the server and the time it takes to invoke a request on the server. The dependence becomes significant with thousands of objects handled.
- The dependency between the number of objects and invocation times is less dramatic on the Pentium Pro machine than it is on others we have had the chance to test.



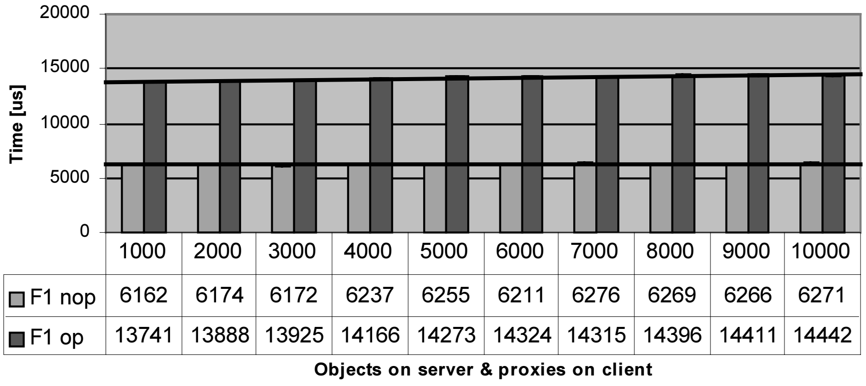
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ Get	5035	6721	8902	11069	13059	15226	16876	18920	21133	23222

Fig. 30: *PROXY.LOT* NT AMD200 local



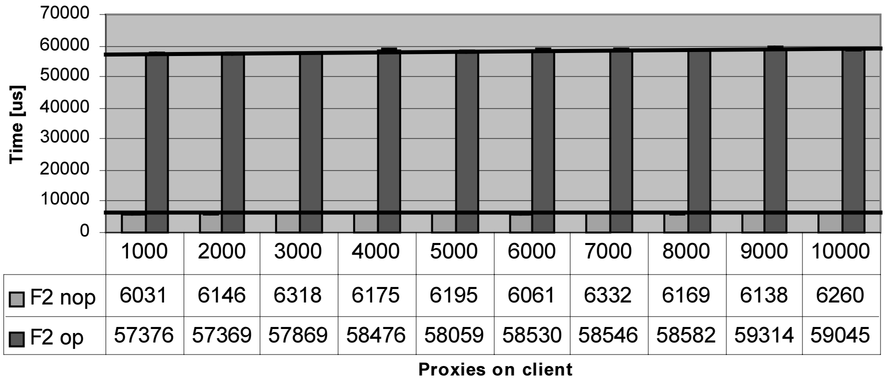
	1000	2000	3000	4000	5000
▨ Get	3286	3509	3947	4272	4960

Fig. 31: *PROXY.LOT* NT PentiumPro Int179 local



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ F1 nop	6162	6174	6172	6237	6255	6211	6276	6269	6266	6271
■ F1 op	13741	13888	13925	14166	14273	14324	14315	14396	14411	14442

Fig. 32: *PROXY.LOT* NT AMD200 local



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ F2 nop	6031	6146	6318	6175	6195	6061	6332	6169	6138	6260
■ F2 op	57376	57369	57869	58476	58059	58530	58546	58582	59314	59045

Fig. 33: *PROXY.LOT* NT AMD200 local

7.1.3.2 Resources vs. Number of Objects

For 10000 objects, the client consumes 1232 bytes per object proxy, the server consumes 1141 bytes per object (a trivial implementation object with no attributes and one operation `void x()`).

7.1.3.3 Resources vs. Queued Requests

7.1.3.3.1 ONEWAY

When testing Orbix 2.2c01, the number of oneway void calls that could be issued before the client blocked was only limited by machine resources. The calls were queued until all available virtual memory was consumed (in our tests, 30000 queued oneway calls consumed 100MB RAM).

Details can be found in the Orbix C++ knowledge base as articles number 138.96, 309.556, and 600.714.

7.1.4 Robustness

No unexpected crashes and no bugs were encountered in Orbix under normal usage conditions. The ORB exhibited a stable performance in our tests.

7.1.4.1 Reliable Servers

Reliable servers can be constructed using the auto-rebinding feature and the Orbix daemon (daemon used for registering the object implementations and for dynamic activation of the object implementations). Persistent objects (based on Loaders) can be used as well—after a server crash, the Orbix daemon will restart the server, the object state can then be restored using Loaders.

7.1.4.2 Limits

7.1.4.2.1 SIZE

In Orbix, the maximum request size is not limited (tested up to 32MB requests). When a too large message arrives at the server, the machine resources (memory) are exhausted, no exception is raised.

7.1.4.2.2 Number of Objects

Orbix 2.2MT AIX server terminates after 35500 objects are created ("Server killed"). On the NT platform, the number of objects was only limited by the amount of available memory (as far as we could test, the AIX limits do not apply on NT).

7.1.4.3 Bugs

7.1.4.3.1 Server name case sensitivity bug

Whenever a server is started, it checks whether it has been registered by putit. Manually launched servers use a case-insensitive check, automatically launched servers use a case-sensitive check. Hence, a server may fail to start (or re-start) automatically even when it can be launched manually. This behavior is exhibited by Orbix 2.2c01 NT and on Orbix 2.3c NT.

7.1.4.3.2 Module level typedefs bug

The Orbix IDL compiler generates incorrect code e.g. for "`module A { typedef sequence <int> B; };`" (the generated C++ code cannot be compiled because of undefined static functions). This is a bug of the Orbix 2.2c01 IDL compiler, and is fixed in Orbix 2.3c.

7.1.4.3.3 Scoped name collision bug

The Orbix IDL compiler generates an error message when two names in different scopes differ in case only (e.g. for "`interface A { void a (); };`"). This is a bug of the Orbix 2.2c01 IDL compiler, and is fixed in Orbix 2.3c.

7.1.4.3.4 User name with spaces bug

Orbix 2.2c01 daemon has problems handling user account names that contain spaces (among other things, rights to launch and invoke servers are not tested correctly).

7.1.4.3.5 Telnet to IIOP port bug

Orbix 2.2c01 daemon crashes when someone telnets to its IIOP listening port and starts sending garbage. This bug is fixed in Orbix 2.3c.

7.1.4.3.6 Interoperability Bug

Orbix server on NT cannot be accessed by other than Orbix client. Orbix 2.2 server on AIX has problems with sequences. For more information see the sections 7.1.1.4 and 9.1.1.2.

7.2 Services

Of the standardized services, the Naming Service (OrbixNames), the Events Service (OrbixEvents), the Trading Service (OrbixTrader), the Security Service (OrbixSecurity), and the Transaction Service (OrbixOTS) are provided. A number of proprietary services is also provided, including OrbixTalk for messaging (complies with but significantly extends the Events Service), OrbixSSL for secure connections, and a number of adapters to products such as ObjectStore, Versant, and MQSeries.

7.2.1.1 Lifecycle Service

Neither the standardized Lifecycle Service nor any functionally similar replacement is provided by Orbix. Reference counting is used to keep track of object and proxy usage and to delete unused objects and proxies.

7.2.1.2 Naming Service

The implementation of the Naming Service in Orbix is called OrbixNames. OrbixNames claims full compliancy with the OMG Naming Service specification and provides all interfaces defined by the standard including the names library. OrbixNames extends the standard by making it possible to bind several objects to a single name.

The OrbixNames server is persistent—the database of name-to-object bindings is preserved across server invocations. This increases the overall robustness of the system that uses OrbixNames.

The OrbixNames server extends the OMG Naming Service standard by providing a special handling of object groups (object group is an object collection implemented by the OrbixNames server). When resolving a name bound to an object group, the OrbixNames server returns a reference of a single group member, picked using either a round-robin or a random choice algorithm. Using this mechanism, multiple servers can offer a service under the same name (e.g. for load balancing purposes), with clients being unaware of the service replication.

As a part of the service, a collection of command line utilities is provided. The utilities allow service clients to manipulate the contents of the name-to-object bindings database using operations similar to those exported by the standardized IDL interfaces.

7.2.1.3 Trading Service

The implementation of the Trading Service for Orbix is called OrbixTrader. OrbixTrader complies with the OMG Trading Service standard and implements the Linked Trader service (the Linked Trader service functionality includes registering and querying of offers, administering and linking of Traders, but does not include proxy offers).

The OrbixTrader server is persistent—the database of offered services is preserved across server invocations. This increases the overall robustness of the system that uses OrbixTrader.

As a part of the OrbixTrader service, a GUI utility to manage the Trader servers is provided. The utility allows to manage service offers (query, list, add, modify, remove), manage service types (list, add, remove), and to set federation import attributes.

7.2.1.4 Events Service

There are two implementations of the Events Service for Orbix—OrbixEvents and OrbixTalk. OrbixEvents complies with the OMG Events Service specification and provides untyped and typed events using the push communication model on a transient event channel. OrbixTalk complies with the OMG Events Service specification and provides untyped events using the push and pull communication models on a reliable persistent event channel.

The OrbixEvents service comes as two servers, one implementing the untyped events, the other implementing the typed events. The untyped events server can provide an arbitrary number of event channels and can buffer pending events in a transient storage. The typed events server provides a single event channel.

In the OrbixTalk service, events are grouped hierarchically using Uniform Resource Locator (URL) like topic names (managed by the OrbixTalk Directory Enquiries daemon). Compared to the Naming Service, this mechanism is more flexible, as the event consumers can select their topics of interest using wildcards.

The OrbixTalk service implements the event delivery mechanism atop the UDP/IP multicast support, thus yielding considerable scalability with respect to the number of attached consumers. Multicast IP addresses are assigned on a per-topic basis by the OrbixTalk Directory Enquiries daemon.

By itself, the OrbixTalk reliable multicast protocol guarantees reliable (ordered) delivery (all consumers receive the same events in the same order). In addition to this, OrbixTalk MessageStore event channel can be used to provide persistent logging of events (using the persistent event log, a consumer can request past events to be replayed e.g. during a failure recovery process).

As a part of the OrbixTalk service, a collection of utilities is provided. The utilities allow service clients to analyze the state logs of the event suppliers and consumers, to dump the contents of the persistent event channels, and to monitor the OrbixTalk multicast network traffic.

7.2.1.5 Persistency Service

All by itself, Orbix supports persistent object references and launching servers on demand. This allows persistent servers to be implemented with significant amount of coding. To automate the additional coding related to providing persistency, and to provide for integration of persistent servers with existing database products, the Orbix Database Adapter, Orbix+ObjectStore Adapter, and the Orbix+Versant Adapter are provided.

The Orbix Database Adapter is a generic framework whose purpose is to help server implementors integrate an arbitrary database with the Orbix ORB. The adapter helps to implement all basic tasks related to providing persistent objects (object reference generation, activation, registration, operation dispatching and transaction management). A detailed documentation with a step-by-step adapter implementation guide and a diagnostic support are provided as a part of the Orbix Database Adapter.

The Orbix+ObjectStore Adapter and Orbix+Versant Adapter are tailor-made to integrate Orbix with specific object-oriented databases (ObjectStore and Versant). Both adapters are similar in structure to the generic Orbix Database Adapter and allow the server implementor to use the specific ODBMS without having to care about the issues of ODBMS-ORB interaction.

Apart from the three adapters listed above, the OrbixOTS transaction service can be used to synchronize access to database resources through the standardized X/Open XA interface.

7.2.1.6 Transaction Service

The implementation of the Transaction Service for Orbix is called OrbixOTS. OrbixOTS is based on the X/Open distributed transaction processing reference model and can be accessed through both the X/Open TX/XA and the standardized OMG OTS interfaces. The four main components of OrbixOTS are the transaction and resource managers, the concurrency control service, and the logging and recovery mechanism.

The OrbixOTS transaction manager is responsible for coordinating distributed transactions. The transaction manager comes in a form of a library linked to all nodes (both clients and servers) participating in a transaction. The manager supports multi-threading and nested transactions.

The OrbixOTS resource manager provides access to transaction resources. The resource manager supports the X/Open XA interface, making it possible to connect products such as Oracle, Sybase or Informix RDBMS to the transaction service in addition to the OTS resource objects.

The OrbixOTS concurrency control service is a straightforward implementation of the OMG Concurrency Control Service. Applications that require locking can be linked with this service and lock objects through its interfaces.

The OrbixOTS logging provides a persistent log of transactional operations. The log can be stored in a file or on a raw device and optionally mirrored. OrbixOTS uses the log to replay operations during crash recovery.

As a part of the OrbixOTS service, a utility to view the status of past and running transactions, to commit and abort running transactions, and to manage transaction logs is provided.

7.2.1.7 Security Service

An implementation of the OMG Security Service is provided (OrbixSecurity), as well as an interface to the SSL layer (OrbixSSL).

7.3 Miscellanea

7.3.1 Requirements

Orbix support a very wide range of platforms, from PCs to mainframes and real-time systems:

- Windows NT/95 (including MT and OCX support) / version 2.x
- Sun Solaris 2.x (ST/MT) / version 2.3
- Hewlett Packard HP/UX 10.20 (ST/MT) / version 2.3
- Hewlett Packard HP/UX 11.0 (MT), (ANSI C++ Compiler) / version 2.3
- Silicon Graphics IRIX 6.x / version 2.x
- IBM AIX 4.2.1 (MT) / version 2.3
- Digital Unix 4.0 (MT) / version 2.3
- Digital Alpha/Open VMSv7.1 / version 2.2
- Sequent SVR4 / version 2.01
- MVS / version 2.x
- VxWorks 5.3 / M. 68K, PowerPC (NT-Hosted, Solaris-Hosted) / version 1.3.5
- QNX 4 / version 2.2

7.3.2 Development Support

7.3.2.1 Documentation

The Programmer's Guide, the Programmer's Reference Guide, the Installation and Administration Guide, and the ActiveX Integration Guide (for Windows versions of Orbix) are shipped with Orbix (both printed and in the Adobe Acrobat PDF format). The guides are well organized and readable.

In addition to the documentation, IONA maintains a Knowledge Base of technical articles related to its products. The Knowledge Base is well organized and many frequent questions and undocumented features are mentioned there. The Knowledge Base is freely accessible from the IONA homepage (http://www.iona.com/support/kb). The IONA home page also offers a collection of design patterns for Orbix and OrbixWeb (called Cookbook and Internet Cookbook).

7.3.2.2 Development Support

Feature		
Access to the reference count value	**Yes**	`CORBA::Object::_refCount()`
Warn when negative reference count	No	
Range checking on sequences	No	the [] operator does not check bounds
Generation of skeleton implementation	**Yes**	IDL compiler switch "`-S`"
Symbolic exception dumping to stream	**Yes**	overloaded operator << for exception type
Broker activity logging	**Yes**	
Reference parsing utilities	No	undocumented call `makeIOR()`

7.3.2.3 Technical Support

IONA offers technical support for all its products. A customer needs to buy a special support contract to obtain full technical support (however, the technical support staff also answered our questions even when we had no such contract, albeit the response times were sometimes long). Patches for Orbix can be downloaded from the IONA web page[1].

7.3.3 Vendor

7.3.3.1 Pricing Policy

The following table lists IONA products for Windows 95 and NT as advertised by IONA on 13 May 1998. The prices are in US dollars.

[1] http://www.iona.com/support/rolled-up/index.html

Product	Type	Price (US$)
Orbix	Development	2995
Orbix	Runtime	2500
OrbixWeb (Professional)	Development	1499
OrbixWeb	Development	799
OrbixNames	Development	495
OrbixTrader	Runtime (Primary CPU)	5000
OrbixTrader	Runtime (Secondary CPU)	2500
OrbixEvents	Runtime (Primary CPU)	1500
OrbixEvents	Runtime (Secondary CPU)	750
OrbixTalk	Development	2500
OrbixManager	Development	1500
Orbix DB Adapter Framework	Development	1000
OrbixOTM	Development	6500
OrbixOTM	Runtime (Primary CPU)	4000
OrbixOTM	Runtime (Secondary CPU)	2000
OrbixSecurity	Development	2500
OrbixSecurity	Runtime	5000
OrbixOTM	Support	975
all other products	Support	650

IONA provides discounts for commercial, inhouse and academic projects. These start at 25% and are project based. Arrangements can be made if a number of products is purchased.

7.3.3.2 Company

IONA Technologies PLC, originally based in Ireland (now with offices in Australia, Germany, Great Britain, Hong Kong, USA), was founded in 1991. The company specializes in CORBA technology and CORBAservices. The products include a wide range of implementations of various OMG specifications.

8 Inprise VisiBroker

Broker:	**VisiBroker**
Vendor:	**Inprise Corp.**
Version tested:	**VisiBroker for C++ 3.0 on Windows NT, AIX**
Platforms used:	**Windows NT 4.0 Workstation** ▪ **200MHz AMD K5 64MB RAM** ▪ **166MHz Intel Pentium 64MB RAM** ▪ **179MHz Intel PentiumPro 64MB RAM** **AIX 4.1 on IBM RS6000-595** **network: 10BASE-T Ethernet** **compilers:** ▪ **Microsoft Visual C++ 4.2** ▪ **IBM C Set ++ 3**

VisiBroker is a CORBA 2.0 ORB now distributed by the Inprise Corporation. Recently, the ORB was distributed by Visigenic, and is based on the code of Orbeline (PostModern Computing.) The ORB is provided for several C++ and Java platforms, accompanying products (e.g. object services) are also available.

8.1 Object Request Broker

8.1.1 Standardized Functionality

8.1.1.1 Interface Definition Language

VisiBroker IDL conforms to the CORBA 2.0 specification. The mapping to C++ conforms to the specification. VisiBroker 3.2 (not tested in this report) provides some of the newer IDL extensions (e.g. the DoubleDouble, WString, WChar types).

8.1.1.2 Basic Remote Invocation

VisiBroker is a full implementation of the CORBA 2.0 standard. It provides complete ORB, BOA, DII, DSI, Interface Repository (IR) and Implementation Repository. VisiBroker supports the invocation semantics defined by the CORBA standard (at-most-once and best-effort), allows implementing server activation policies, and supports exceptions and context handling.

The semantics of the `BOA::obj_is_ready()` call differs from the standard. With VisiBroker, the `obj_is_ready()` call is used to register individual object implementations on the server (but the ORB also has an undocumented automatic registering feature). For more information about object creation and deletion, see section 8.1.2.1.3.

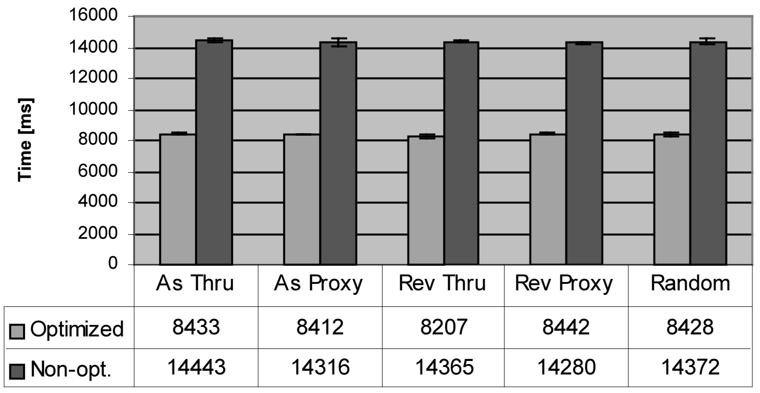
VisiBroker supports both the "at-most-once" and "best-effort" invocation semantics. In the default configuration, however, the “at-most-once” semantics does not conform to the specification, which says that an operation should be invoked exactly once or an exception should be raised. VisiBroker provides automatic reconnection regardless the completition status of the invocation (see section 8.1.2.1.6).

Dynamic activation policies are supported using Implementation Repository and Object Activation Daemon. All basic activation policies described by the standard are supported.

8.1.1.2.1 Measurements Verification

CALL

The results of the CALL benchmark indicate there is no significant dependence between the invocation times and the object invocation pattern. This means that the results of the other benchmarks can be applied to applications with different invocation patterns as well.



	As Thru	As Proxy	Rev Thru	Rev Proxy	Random
▨ Optimized	8433	8412	8207	8442	8428
■ Non-opt.	14443	14316	14365	14280	14372

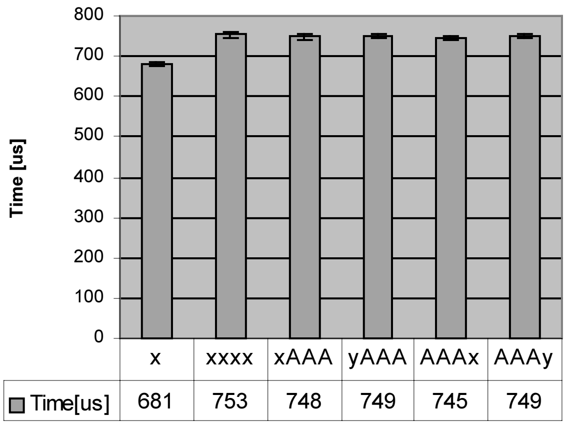
Fig. 34: *CALL* NT Int179 local (10.000 calls)

8.1.1.2.2 Dispatcher Performance

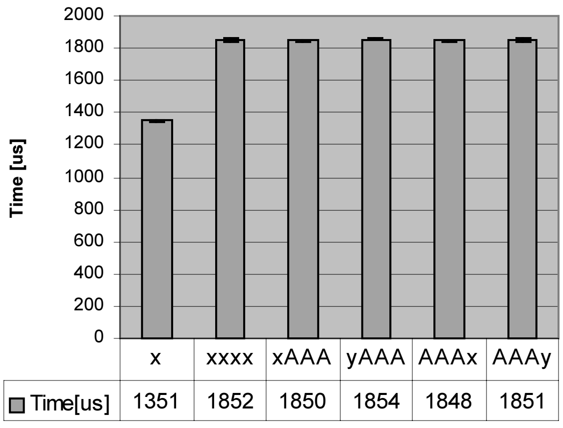
IDL.NAMES

The most important result from the IDL.NAMES benchmark is the demonstration of the fact that calling operations with longer (500 characters) names takes longer time to complete. This is due to the fact that the marshalling, unmarshalling, transport and dispatching algorithms all use a plain string form of the operation names. The test shows that:

- For normal-length names (10-30 characters), the difference in invocation time is insignificant (about 1%).
- The difference in invocation time is more significant when running over a network (this is due to the fact that the operation name is transported in a plain string form).



	x	xxxx	xAAA	yAAA	AAAx	AAAy
▨ Time[us]	681	753	748	749	745	749



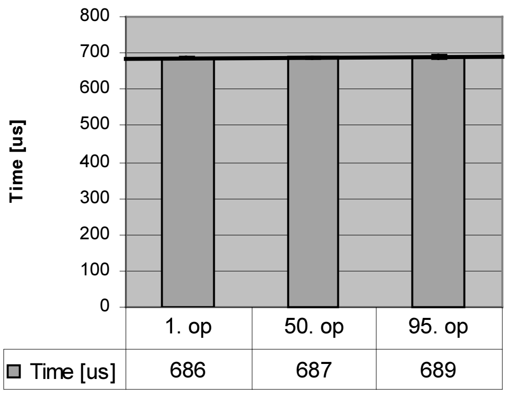
	x	xxxx	xAAA	yAAA	AAAx	AAAy
▨ Time[us]	1351	1852	1850	1854	1848	1851

Fig. 35: *IDL.NAMES* NT AMD200 (1 call)

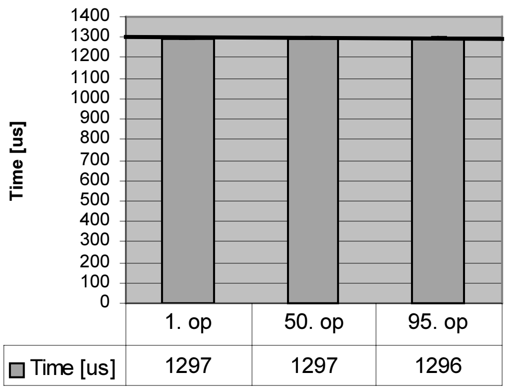
Fig. 36: *IDL.NAMES* NT Int166 -> AMD200 (idle net, 1 call)

IDL.LONG

The relationship between the position of an operation within an interface and the time it takes to complete the operation is not exhibited by VisiBroker.



	1. op	50. op	95. op
▨ Time [us]	686	687	689



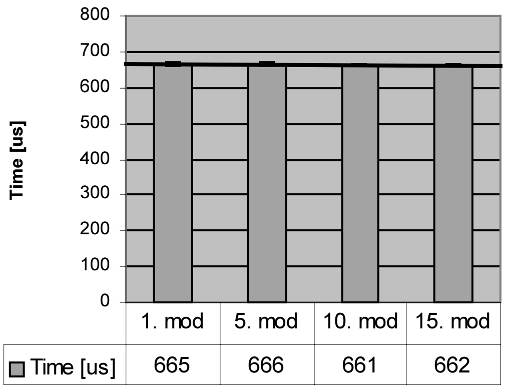
	1. op	50. op	95. op
▨ Time [us]	1297	1297	1296

Fig. 37: *IDL.LONG* NT AMD200 (1 call)

Fig. 38: *IDL.LONG* NT Int166 -> AMD200 (idle net, 1 call)

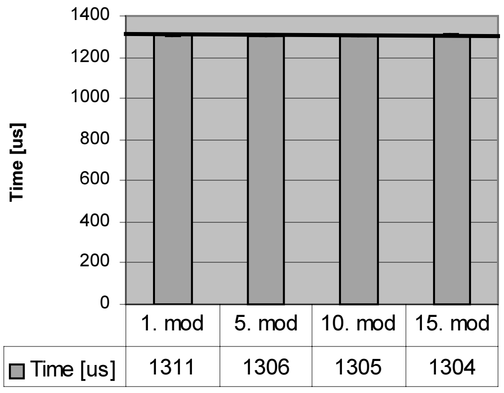
IDL.DEEP

This benchmark indicates there is no relationship between the interface nesting depth and the time it takes to complete an operation of that interface.



	1. mod	5. mod	10. mod	15. mod
▨ Time [us]	665	666	661	662

Fig. 39: *IDL.DEEP* NT AMD200 (1 call)



	1. mod	5. mod	10. mod	15. mod
▨ Time [us]	1311	1306	1305	1304

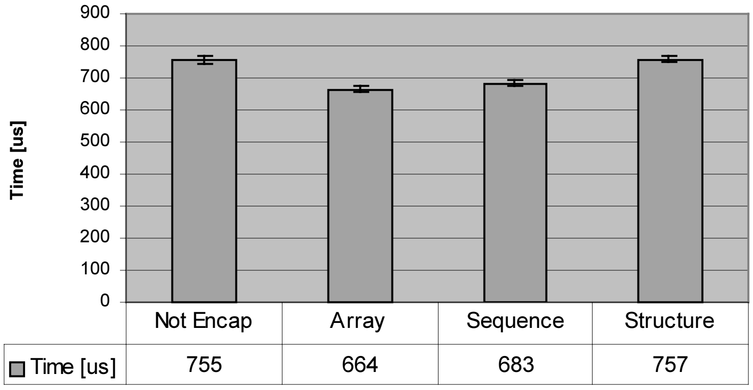
Fig. 40: *IDL.DEEP* NT Int166 -> AMD200 (idle net, 1 call)

8.1.1.2.3 Overall Performance

IDL.ENCAP

The IDL.ENCAP benchmark compares the invocation time when passing 100 longs individually, in an array, in a bounded sequence, and in a structure. Overall, the tests show that:

- The deviation in invocation times is about 10%, with arrays having the fastest and individual longs the slowest invocation time.



	Not Encap	Array	Sequence	Structure
▨ Time [us]	755	664	683	757

Fig. 41: *IDL.ENCAP* NT AMD200

THROUGH.IN

The tests show that:

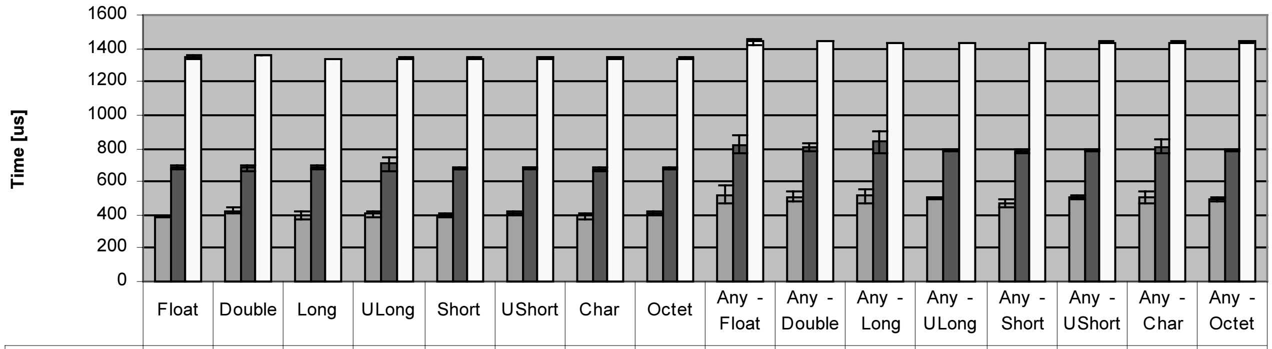
- Time to pass a basic IDL data type stays roughly the same regardless of the type being passed (deviation is about 10%).
- Time to pass an array or a sequence of a basic IDL data type depends mostly on the length of data in octets.

- Encapsulating an argument within any adds a substantial overhead. Hence, when passing an array or a sequence of anys, the invocation time primarily depends on the number of anys.

Running the same test suite over an idle 10BASE-T Ethernet shows that:

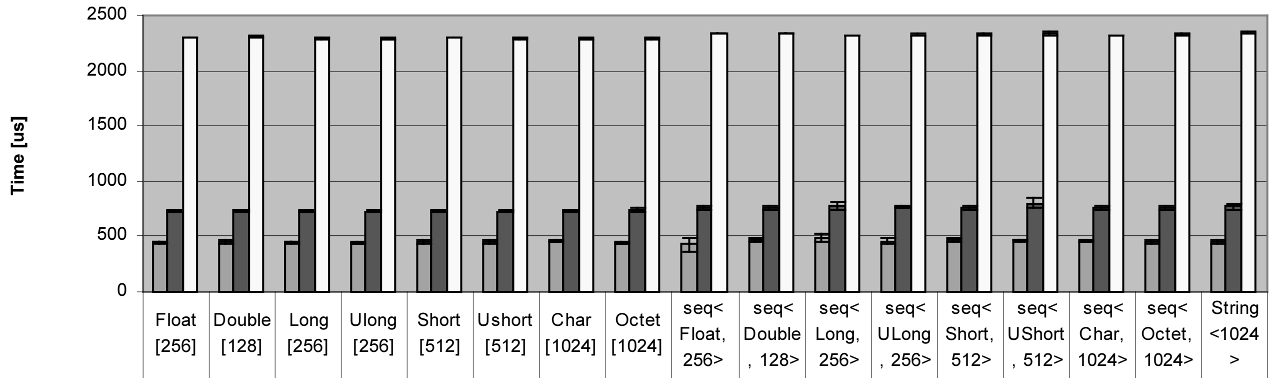
- Unless a very large number of complex arguments is passed, the constant overhead of an invocation overshadows the impact of argument sizes and types. Hence, it pays off to call less often with more arguments than vice versa.

- Local roundtrip times are roughly 165% of local delivery times (except for any data types). Remote RTT is roughly 188% of local RTT for basic data types, 310% of local RTT for arrays, 300% of local RTT for sequences (note that the local measurements use an optimized IPC mechanism).



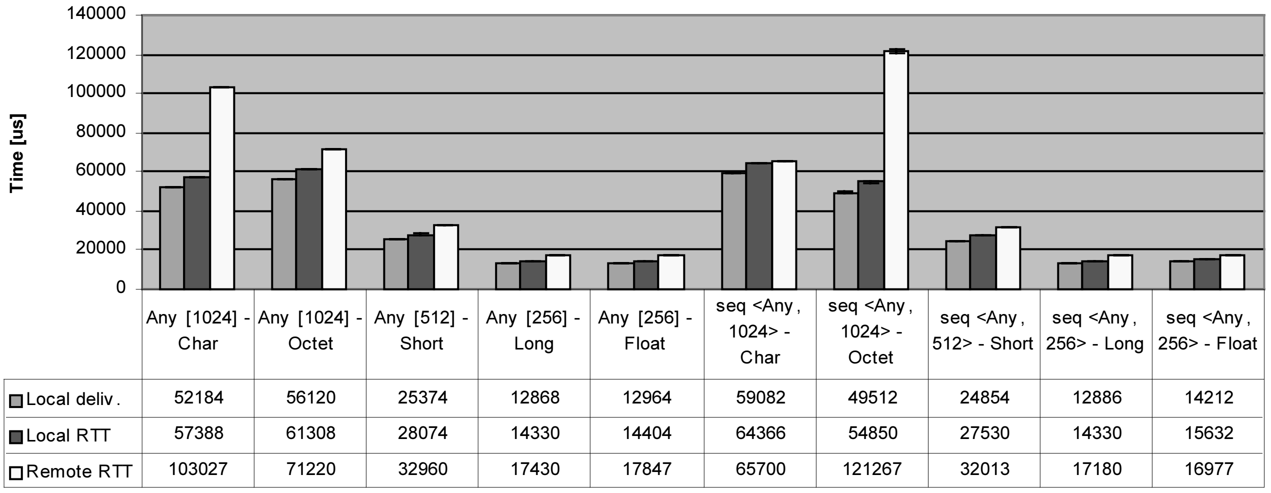
	Float	Double	Long	ULong	Short	UShort	Char	Octet	Any - Float	Any - Double	Any - Long	Any - ULong	Any - Short	Any - UShort	Any - Char	Any - Octet
▨ Local deliv.	392	422	396	406	398	404	392	408	522	508	514	500	470	500	510	494
■ Local RTT	686	682	686	706	678	680	672	676	824	810	838	792	784	788	812	788
□ Remote RTT	1347	1360	1340	1337	1337	1343	1343	1340	1443	1443	1433	1433	1433	1437	1437	1440

Fig. 42: *THROUGH.IN* NT AMD200, Int166->AMD200 idle net (time to pass **basic** data types—1 call)



	Float [256]	Double [128]	Long [256]	Ulong [256]	Short [512]	Ushort [512]	Char [1024]	Octet [1024]	seq< Float, 256>	seq< Double , 128>	seq< Long, 256>	seq< ULong , 256>	seq< Short, 512>	seq< UShort , 512>	seq< Char, 1024>	seq< Octet, 1024>	String <1024 >
▨ Local deliv.	444	452	444	442	448	454	460	442	432	472	496	456	464	464	456	452	452
■ Local RTT	732	736	732	730	732	732	738	742	764	766	780	766	764	802	760	762	772
□ Remote RTT	2297	2303	2293	2290	2300	2293	2290	2293	2337	2330	2323	2327	2330	2337	2323	2333	2343

Fig. 43: *THROUGH.IN* NT AMD200, Int166->AMD200 idle net (time to pass **compound** data types—1 call)



	Any [1024] - Char	Any [1024] - Octet	Any [512] - Short	Any [256] - Long	Any [256] - Float	seq <Any, 1024> - Char	seq <Any, 1024> - Octet	seq <Any, 512> - Short	seq <Any, 256> - Long	seq <Any, 256> - Float
▨ Local deliv.	52184	56120	25374	12868	12964	59082	49512	24854	12886	14212
■ Local RTT	57388	61308	28074	14330	14404	64366	54850	27530	14330	15632
□ Remote RTT	103027	71220	32960	17430	17847	65700	121267	32013	17180	16977

Fig. 44: *THROUGH.IN* NT AMD200, Int166->AMD200 idle net (time to pass **any** data types—1 call)

8.1.1.3 Dynamic Type Manipulation

VisiBroker fully supports DII and DSI, the DII mechanism does not work when server and client are collocated though. VisiBroker 3.2 (not tested in this report) supports the DynAny type as well.

8.1.1.4 Repositories

The Interface Repository is provided as a standalone server. The Implementation Repository cooperates with a special daemon used for activating servers (OAD).

8.1.1.5 Interoperability

VisiBroker's native protocol is IIOP, the interoperability tests did not show any problem with the IIOP implementation. For more information see section 9.1.1.2.

The VisiBridge bridge should allow OLE applications to connect to a CORBA server. Right now, however, the bridge is not on the Inprise product list.

IIOP can be used to communicate with Java applications. Inprise provides a Java based CORBA implementation called VisiBroker for Java (VisiBroker for Java uses the Caffeine technology).

8.1.2 Nonstandard Extensions

8.1.2.1 Communication Extensions

8.1.2.1.1 Locating Objects

To locate object implementations, VisiBroker provides a proprietary mechanism called the Location Service. The Location Service can be queried for all available instances of an object (or their description) based on the interface repository ID or (more precisely) on the ID coupled with the instance name.

Central to the Location Service is a specialized daemon (Smart Agent). A single Smart Agent is capable of locating all object implementations within a single network segment (but more Smart Agents can be running within a single segment as well). Smart Agents from different networks can exchange information about available object implementations.

The object implementations are registered automatically with the Smart Agent. Locating an implementation can either be fully transparent or controlled by the application programmer. Apart from submitting queries to the Location Service, the programmer can also register triggers (triggers notify the client of changes in the availability of object instances).

8.1.2.1.2 Binding To Objects

The `_bind()` call in VisiBroker allows the client side code to bind to an object of a given interface even when the server or the object key is not specified. The actual binding can be deferred until the first request is sent to the object.

8.1.2.1.3 Instantiating Implementations

To create an object implementation accessible from outside of the server address space, the following steps need to be performed:

(1) create new object, passing an instance name to it (`obj = new XXX_i ("name")`)

(2) register the object using the `BOA::obj_is_ready()` function (`BOA->obj_is_ready(obj)`)

(3) at this point, the object reference can be exported[1]

To delete the object implementation, the following step should be done:

- unregister the object using the `BOA::deactivate_obj()` call
- decrease the object's reference count to zero using `release()`

The `BOA::obj_is_ready()` call (normally used to tell BOA the server is ready to handle requests in the unshared activation mode) has a non-standard semantics in VisiBroker. It should be called on all CORBA objects to notify the ORB of their existence.

8.1.2.1.4 Persistent References

In VisiBroker, persistent object references are provided by a mechanism based on Activators. Activators are used for deferring object activation until a client request, a single Activator can serve a single object or a group of objects.

8.1.2.1.5 Automatic Activation

The Object Activation Daemon is able to launch a server automatically whenever a request for the server comes in and the server is not running. The VisiBroker 3.0 C++ NT automatic activation has been tested and works OK.

8.1.2.1.6 Automatic Reconnection

Lost connections are re-established automatically regardless of the completion status of the call, which is dangerous in most cases (in case of failure, operation invocation can be lost, or duplicated). This (default) behavior can be altered

---

[1] As an undocumented feature, VisiBroker automatically registers objects whose references are to be passed outside the server address space.

using either client side interceptors, an additional argument to `_bind()`, or a process-wide bind defaults setting (this affects the `string_to_object()` behavior as well).

8.1.2.1.7 Invocation Extensions

The client side proxy object can be customized in VisiBroker. This allows the client to change the default invocation semantics and introduce optimizations such as caching when needed. The default proxy object is usually reused to provide the communication support.

When the client and the server reside in different processes on the same host, VisiBroker uses shared memory (on NTs) or UNIX Domain Protocol (on UNIXes) for data transport. According to our measurements, this speeds up the communication about twice on a 200MHz Pentium running Windows NT.

Callbacks can easily be implemented using the VisiBroker multi-threading support.

8.1.2.1.8 Customizability

In VisiBroker, various events can be hooked using Interceptors. Interceptors can monitor or alter communication between the client and the server. On the client side, Interceptors are installed on a per-object basis, on the server side on a per-connection basis (a single client interceptor can be created to handle all server requests, a single server interceptor can be created to handle all client connections).

Interceptors can control access to individual methods, view, add or modify the transmitted data, dynamically select a server that receives the invocation, or provide an alternative location service should the default binding fail. The Interceptors are separated into three groups:

- *binding interceptor* - notify when bind is called, bind succeeded, bind fails, the same for rebind, when exception during bind occurred
- *client interceptor* - notify when preparing request, sending request, sending failed, sending succeeded, when receiving reply, receiving failed, when exception occurred
- *server interceptor* - notify when locating, locate forwarded, locate failed, locate succeeded, receiving request, preparing reply, sending reply, sending failed, request completed, when exception occurred

More than one Interceptor can be installed. The Interceptors are chained in order of their registration with the ORB. Interceptors can also be installed when the client and server are collocated in the same address space (but a special construct is needed in this case).

VisiBroker allows to integrate its event processing loop with the application event handler (this is useful especially in single-threaded applications). Ready-made dispatcher classes for Windows and X Windows environments are provided. An application programmer can write his own dispatcher class for other event-driven environments.

8.1.2.2 Multi-threading Extensions

Two sets of libraries (multi-threaded as well as single-threaded) are shipped with VisiBroker. Both sets provide a compatible interface, thus an application code written for use with the single-threaded libraries should execute correctly if linked with the multi-threaded libraries.

The multi-threaded libraries are thread safe and re-entrant. VisiBroker allows for both the client and the server to be multi-threaded. Multi-threaded and single-threaded clients and servers can be combined in any fashion.

8.1.2.2.1 Multi-threaded Servers

At the server side, VisiBroker directly provides three of the threading strategies listed in chapter 3.2.2.1:

- *single threaded* - when linked with the single-threaded version of the VisiBroker ORB library
- *pool of threads* - default when linked with the multi-threaded version of the VisiBroker ORB library
- *thread per session*

Each of these strategies is implemented as a specific Basic Object Adapter. The selection of a specific threading strategy is straightforward—the programmer specifies the strategy as an argument to the `BOA_init()` call (the third argument of the call is a string (`char *`) and can be `"TSingle"`, `"TPool"` or `"TSession"`). The strategy can also be set on a command-line when launching the server application using the `-OAid` prefix (e.g. `server -OAid TPool` will set the server to use the thread pooling strategy).

When using the TPool or TSession strategies, the server has to be linked with the multi-threaded version of libraries[1].

The maximum number of threads in a pool can be set using the `BOA::thread_max()` call[2]. This call also has a command-line equivalent (e.g. `server -threadMax 20` will create a pool of 20 threads).

It is difficult to implement a proprietary multi-threading strategy (different from the three strategies automatically supported by the ORB), as the ORB provides virtually no support to do this.

8.1.2.2.2 Multi-threaded Clients

A VisiBroker client can be multi-threaded, the client-side ORB libraries are thread-safe. By default, all client threads share the same TCP/IP connection to the server. This default behavior can be changed by calling the `_clone()` operation on the remote object's proxy (this creates a new proxy, this proxy then opens a new TCP/IP connection to the server).

Both the MT.CLIENT and the MT.CONN benchmarks take 50 seconds to complete, indicating that VisiBroker clients can issue parallel requests both over single and multiple connections

8.1.2.2.3 Multi-threaded ORB

At the ORB level, a single TCP/IP connection is used (by default) to transmit requests between two address spaces. This default behavior can be changed by the client side code as described above. The connection management in VisiBroker is flexible, the maximum number of connections can be limited at both the client and the server sides.

To manage threads, VisiBroker provides the `thread_max()` and `thread_stack_size()` calls (both for setting and retrieving the current value). For connection management, VisiBroker provides `connection_max()` for setting and getting the maximum number of concurrent connections and `connection_count()` for getting the current number of connections.

---

[1] Surprisingly, the multi-threaded libraries do not work when using the `TSingle` strategy, executable file cannot run (an error occurs during initialization).

[2] However, the benchmarks indicate that limiting the maximum number of threads with the `BOA::thread_max()` call does not work when using the locally optimized communication mechanism.

8.1.2.2.4 Concurrency

Both at the client and at the server side, it is the application programmer who is responsible for protecting shared data structures at the application level (using the underlying operating system's synchronization primitives).

8.1.2.3 Management Extensions

The basic VisiBroker installation provides a set of utilities for managing the Implementation Repository (used to implement dynamic activation of servers) and querying the Location Service (used to locate available object implementations). In a separate package, a suite of Java-based management tools (VisiBroker Productivity Tools) is provided. The suite consists of VisiBroker Manager, VisiBroker Performance Monitor and VisiBroker Application Partitioner for Java.

The VisiBroker Manager is a graphical interface for accessing the VisiBroker Naming Service, the Location Service, the Interface Repository, and the Implementation Repository. The manager offers an alternative to the command line utilities provided with VisiBroker.

The VisiBroker Performance Monitor displays performance data related to various aspects of a distributed application execution. The monitor provides a hierarchical view of all monitorable servers, each with a list of the implemented objects and operations (a monitorable server must be linked with a special library). The monitor can provide information on specific operations, measure latency of calls and the input and output buffer sizes for argument lists (minimum, maximum and average values). The performance data collected by the monitor are also accessible through both IDL and ActiveX interfaces.

8.1.3 Scalability

8.1.3.1 Speed vs. Number of Objects

8.1.3.1.1 PROXY

The PROXY.CREATE benchmark indicates there is no dependence between the number of objects and the object creation time both on the server and the client sides. The PROXY.FIRST and PROXY.NEXT benchmarks show that:

- There is a significant difference between the time it takes to invoke an operation for the first time and for all subsequent times.

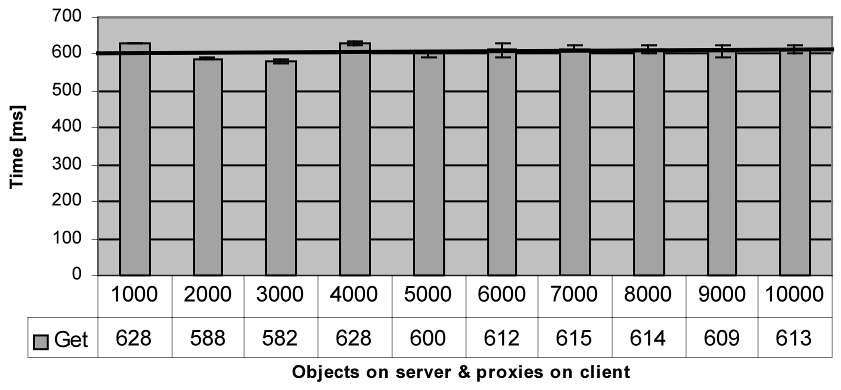
PROXY	**NT local**			**NT idle net**		
objects	1000	2000	3000	1000	2000	3000
CREATE	533	1039	1563	791	1467	2191
FIRST	4054	8035	11822	6208	12340	18931
NEXT	660	1424	1964	1321	2683	3960

Fig. 45: Table for *PROXY* test (time in ms)

8.1.3.1.2 PROXY.LOT

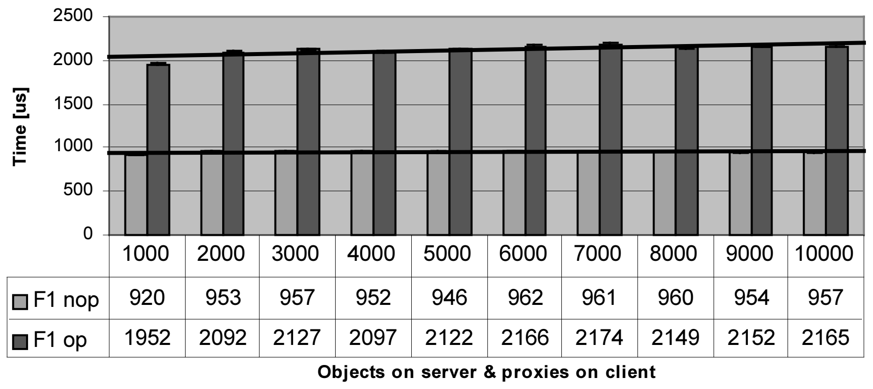
The PROXY.LOT benchmark shows that:

- There is no measurable dependence between the number of objects handled by the server and the time it takes to create and register a new object.
- There is no measurable dependence between the number of objects handled by the client and the time it takes to receive a new object reference.
- There is a linear dependence between the number of objects handled by the server and the time it takes to invoke a request on the server. The dependence becomes significant with thousands of objects handled.



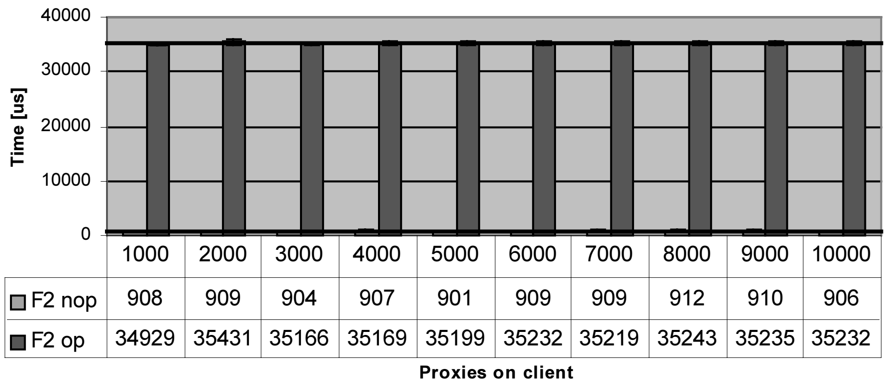
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ Get	628	588	582	628	600	612	615	614	609	613

Fig. 46: *PROXY.LOT* NT Int166



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ F1 nop	920	953	957	952	946	962	961	960	954	957
■ F1 op	1952	2092	2127	2097	2122	2166	2174	2149	2152	2165

Fig. 47: *PROXY.LOT* NT Int166



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▢ F2 nop	908	909	904	907	901	909	909	912	910	906
■ F2 op	34929	35431	35166	35169	35199	35232	35219	35243	35235	35232

Fig. 48: *PROXY.LOT* NT Int166

8.1.3.2 Resources vs. Number of Objects

For 10000 objects, the client consumes 1076 bytes per object proxy, the server consumes 1227 bytes per object (a trivial implementation object with 4 bytes of attributes included).

8.1.3.3 Resources vs. Queued Requests

8.1.3.3.1 ONEWAY

When testing the number of oneway void calls that can be issued to a VisiBroker server before the client blocks, the results were as follows:

- *local client with optimized communication* - the number of oneway calls roughly corresponds to the number of threads available in the server thread pool. When the number of threads in the thread pool is not limited, it will grow until the system resources are exhausted.

- *remote client* - about 350 to 500 plus the size of server thread pool oneway void calls can be issued before the client blocks. Most likely, this is related to network layer buffers. When the number of threads in the thread pool is not limited, it will grow until the system resources are exhausted.

8.1.4 Robustness

No unexpected crashes were encountered in VisiBroker under normal usage conditions, bugs are listed later in this section. The ORB exhibited a stable performance in our tests.

8.1.4.1 Reliable Servers

Reliable servers can be constructed using the auto-rebinding feature and the VisiBroker Object Activation Daemon (a daemon responsible for dynamic activation of the object implementations). Persistent objects are supported as well (using Activators).

VisiBroker object implementations can migrate. To use this feature, the automatic re-binding must be enabled. To achieve migrating of a server with stateless client connections, the implementation must terminate (unregister) itself, migrate (prepare to run at the new host) and start again (register with the daemon used for locating of objects). The clients loose their connections and attempt to re-bind automatically.

On a server that maintains state of client connections, migrating is not transparent to the client (the client must use an interceptor). When the connection to the object implementation fails and the ORB re-connects the client to the new server, the interceptor's `rebind_succeeded()` method will be invoked by the ORB. The client is then supposed to restore the state of the server connection as necessary.

A very simple load balancing is performed by the object location daemon. The daemon sequentially (round robin algorithm) routes connection requests to different object implementations (when asked for the same interface or instance name). To provide a proprietary load balancing mechanism, a programmer can use ORB extensions such as the modified proxies (Smart Stubs), the Location Service (with triggers), and the Interceptors.

8.1.4.2 Limits

8.1.4.2.1 SIZE

By default, VisiBroker does not limit the maximum message size (tested up to 32MB), a message size limit cannot be set by the programmer. When the resources are close to exhaustion, the CORBA::UNKNOWN exception is raised.

8.1.4.2.2 Number of Objects

VisiBroker 3.0 C++ AIX server terminates after 32000 objects are created ("Assertion failed"). On the NT platform, the number of objects was only limited by the amount of available memory (as far as we could test, the AIX limits do not apply on NT).

8.1.4.3 Bugs

8.1.4.3.1 Thread pool size limit bug

The `CORBA::BOA::thread_max()` call (supposed to limit the number of threads in a thread pool) does not work when using the optimized local communication. Tested on VisiBroker 3.0, 3.1, 3.2 C++ NT.

8.1.4.3.2 AIX utilities bug

It was not possible to launch the VisiBroker 3.0 C++ Smart Agent, Object Activation Daemon, and some other utilities on AIX 4.1. This might be due to the fact that VB 3.0 requires AIX 4.2.

8.2 Standardized Services Functionality

Of the standardized services, the Naming Service (VisiBroker Naming Service for C++ 3.0), the Event Service (VisiBroker Event Service for C++ 3.0), and the Transaction Service (TPBroker) are provided. A support for security (VisiBroker Developer SSL Pack for C++ 3.0) is also available.

8.2.1.1 Naming Service

The implementation of the Naming Service in VisiBroker is called VisiBroker Naming Service (VBNS). VBNS claims full compliance with the OMG Naming Service specification and provides all interfaces defined by the standard including the names library.

The VBNS server is persistent—the database of name-to-object bindings is preserved across server invocations. This increases the overall robustness of the system that uses VBNS.

As a part of the service, a utility to manipulate the contents of the name-to-object bindings database using operations similar to those exported by the standardized IDL interfaces is provided. The utility is only available in Java version.

8.2.1.2 Events Service

The implementation of the Events Service for VisiBroker is called VisiBroker Events Service (VBES). VBES claims full compliance with the OMG Events Service specification and provides untyped events using the push and pull communication models on a transient event channel.

VBES is shipped as a separate server and as a library. The service client can either start VBES as a standalone process, or include it with either the supplier or the consumer application.

8.2.1.3 Transaction Service

The implementation of the Transaction Service for VisiBroker is called TPBroker. Currently available is TPBroker 1.1 (integrated with VisiBroker 2.0), soon to be released is TPBroker 3.0 (integrated with VisiBroker 3.2) and a new product called VisiBrokerITS. TPBroker is based upon the OpenTP1 transaction manager from Hitachi and can be accessed through both the X/Open TX/XA and the standardized OMG OTS interfaces. The two main components of TPBroker are the OTS adapter and the transaction manager.

The TPBroker OTS adapter acts as a wrapper for the transaction manager, converting calls issued on the OMG Transaction Service API to the calls on the X/Open TX/XA interfaces and vice versa. The OTS adapter consists of a linkable library and two daemons responsible for driving the commit and recovery protocols.

The TPBroker transaction manager implements the core of the transaction service. The manager consists of a linkable library and a daemon responsible for process monitoring. The manager only supports flat transactions (TPBroker 3.0 supports flat and nested transactions, implicit and explicit context propagation, direct and indirect context management, multi-threading, administrative functions).

As a part of the TPBroker service, a set of utilities to view the status of running transactions and to manipulate resource managers is provided.

8.2.1.4 Persistency Service

There is no implementation of a separate Persistence Service for VisiBroker. All by itself, VisiBroker supports persistent object references and launching servers on demand. This allows persistent servers to be implemented with significant amount of coding. The TPBroker transaction service can be used to synchronize the access to database resources through the standardized X/Open XA interface.

8.2.1.5 Unsupported Services

Neither the standardized Lifecycle nor any functionally similar replacement is provided. Reference counting is used to keep track of object and proxy usage and to delete unused objects and proxies.

Neither the standardized Trading Service nor any functionally similar replacement is provided.

8.3 Miscellanea

8.3.1 Requirements

8.3.1.1 Supported Platforms And Compilers

List of supported platforms for VisiBroker for C++ follows:

- Windows NT/95 / version 3.2
- Sun Solaris 2.5, 2.5.1, 2.6 / version 3.2
- Sun Solaris 2.4 / version 2.1
- SunOS 4.1.4 / version 2.1
- HP-UX 11.00, 10.20 / version 3.0
- HP-UX 10.20, 10.10, 10.01 / version 2.1
- Silicon Graphics IRIX 6.2, 6.3 / version 3.0
- IBM AIX 4.2, 4.1 / version 3.0
- Digital Unix 4.0a,b,c, 3.2f / version 2.1

8.3.2 Development Support

8.3.2.1 Documentation

The Programmer's Guide, the Reference Guide, and the Installation and Administration Guide are delivered with VisiBroker for C++. The guides are readable and mostly understandable, but the structure is rather poor, making routine searches for information difficult.

8.3.2.2 Development Support

Feature		
Access to the reference count value	**Yes**	`CORBA::Object::_ref_count()`
Warn when negative reference count	No	
Range checking on sequences	**Yes**	exception raised
Generation of skeleton implementation	No	
Symbolic exception dumping to stream	**Yes**	overloaded operator << for exception type
Broker activity logging	No	
Reference parsing utilities	**Yes**	the reference parsing utilities were not a part of our installation, but were sent to us upon request by Visigenic

8.3.2.3 Technical Support

The VisiBroker Technical Support team provides the following support levels:

- **Premium Support.** In addition to unlimited person-to-person and electronic access to product information, customer receives responses to high-priority assistance requests. A non-stop beeper option is also available.
- **Standard Support.** Unlimited person-to-person and electronic access to information about our products.
- **Online Support.** All VisiBroker customers, including evaluation customers, are entitled to use our database of frequently asked questions.

8.3.3 Vendor

8.3.3.1 Pricing Policy

Following are the prices sent to us by Visigenic on April 15, 1998 (effective July 22, 1997)[1]:

Product	Type	Price (US$)
VisiBroker for C++ 3.0	Development	1995
VisiBroker ORB for C++ 3.0	Deployment (server-only)	1665
VisiBroker ORB for C++ 3.0	Deployment (node-by-node)	95
VisiBroker Developer for Java 3.0	Development	1995
VisiBroker ORB for Java 3.0	Deployment (server-only)	1665
VisiBroker ORB for Java	Deployment (node-by-node)	95

VisiBroker Naming service for C++ 3.0	Development	295
VisiBroker Naming service for C++ 3.0	Deployment (machine)	165
VisiBroker Event Service for C++ 3.0	Development	545
VisiBroker Event Service for C++ 3.0	Deployment (machine)	335
VisiBroker Developer SSL Pack for C++ 3.0	Development	195
VisiBroker Developer SSL Pack for C++ 3.0	Deployment (server-only)	695
VisiBroker Developer SSL Pack for C++ 3.0	Deployment (node-by-node)	45

8.3.3.2 Company

Right now, the Inprise Corporation is undergoing a major shift towards the distributed enterprise applications. The company has recently changed its name (from Borland International, founded 1983) and acquired the former distributor of VisiBroker (Visigenic). Of the product range, perhaps most know are the Borland development tools (Delphi, JBuilder, C++Builder) and the VisiBroker ORB.

[1] Please note that with recent acquisition of Visigenic by the Inprise Corporation (then Borland International), the prices will most likely be subject to change.

9 Comparison Summary

9.1 Object Request Broker

9.1.1 Standardized Functionality

9.1.1.1 Basic Remote Invocation

9.1.1.1.1 Measurements Verification

CALL

The results of the CALL test indicate there is no significant relationship between the pattern of the calls issued to the server and the times measured by the benchmark. This means that it is possible to apply the results of other benchmarks to applications with different invocation patterns.

9.1.1.1.2 Dispatcher Performance

IDL.NAMES

The most important result from the IDL.NAMES benchmark is the demonstration of the fact that calling operations with longer names takes longer time to complete. This is due to the fact that the marshalling, unmarshalling and dispatching algorithms all use a plain string form of the operation names. The test shows that:

- For normal-length names (10-30 characters), the difference in invocation time is insignificant (below 1%).
- The difference in invocation time is more significant when running over a network (more data is passed).

IDL.LONG

The IDL.LONG benchmark shows that for Orbix and omniORB, the operation that is defined later in the IDL interface takes longer time to complete. The reason for this is that the dispatcher code (generated from the IDL definition) looks up the operation name using a sequential (linear) search. The relationship between the position of an operation within an interface and the time it takes to complete the operation is not exhibited by VisiBroker. The test shows that:

- For medium-sized interfaces (100 operations), the difference in invocation time between the first and the last operation may be significant with omniORB and Orbix (no difference with VisiBroker).
- The difference in invocation time does not change when running over a network (clearly, only server side code matters).

IDL.DEEP

The IDL.DEEP benchmark indicates a relationship between the interface nesting depth and the time it takes to complete an operation of that interface. The test shows that:

- For interfaces deeply nested in modules (10 levels), the difference in invocation time between operations in non-nested and nested interfaces may be significant with Orbix (no difference with omniORB and VisiBroker).

- The difference in invocation time is more significant when running over a network (more data is passed).

9.1.1.1.3 Overall Performance

IDL.ENCAP

The IDL.ENCAP benchmark compares the invocation time when passing 100 longs individually, in an array, in a bounded sequence, and in a structure. Overall, the tests show that:

- The deviation in invocation times is usually within 10%, with arrays having the fastest (except for omniORB where sequences are slightly faster) and individual longs the slowest invocation time.

THROUGH.IN

The results of the local throughput tests indicate the relative speed of the ORBs is (with a negligible number of exceptions) omniORB (fastest), VisiBroker, Orbix (slowest). The tests show that:

- Time to pass a basic IDL data type stays roughly the same regardless of the type being passed. When passing a single basic IDL data type, the relative invocation times are omniORB 100% (520µs), VisiBroker 130%, Orbix 820%.

- Time to pass an array or a sequence of a basic IDL data type depends mostly on the length of data in octets for Orbix and VisiBroker, and on the number of items for omniORB. For a 1kB array of floats, relative invocation times are omniORB 100% (670µs), VisiBroker 110%, Orbix 660%. For a 1kB array of octets, relative invocation times are VisiBroker 100% (740µs), omniORB 120%, Orbix 590%.

- Encapsulating an argument within any adds a substantial overhead. Hence, when passing an array or a sequence of anys, the invocation time primarily depends on the number of anys. For 1024 anys of octets, relative invocation times are omniORB 100% (14600µs), VisiBroker 420%, Orbix 2800%.

Running the same test suite over an idle 10BASE-T Ethernet shows that:

- Unless a very large number of complex arguments is passed, the constant overhead of an invocation overshadows the impact of argument sizes and types. Hence, it pays off to call less often with more arguments than vice versa.

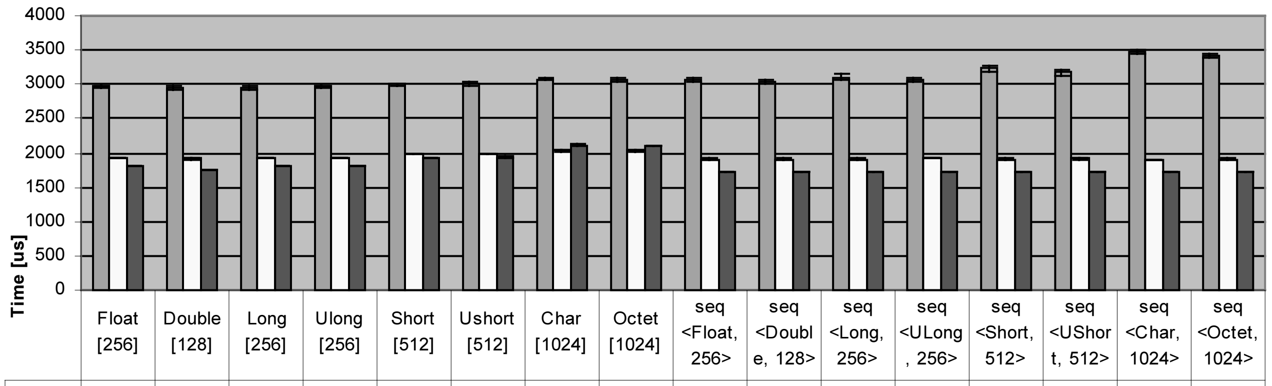
- For basic types, the relative invocation times are omniORB 100% (750µs), VisiBroker 180%, Orbix 590%. For a 1kB array of octets, relative invocation times are omniORB 100% (2100µs), VisiBroker 110%, Orbix 260%. For 1024 anys of octets, relative invocation times are omniORB 100% (12400µs), VisiBroker 570%, Orbix 2700%.

- Paradoxically, local invocation times may sometimes be longer than network invocation times (e.g. when passing arrays of anys in omniORB and Orbix).

With respect to performance, testing of heterogeneous ORB combinations (client and server use different ORBs) indicates that:

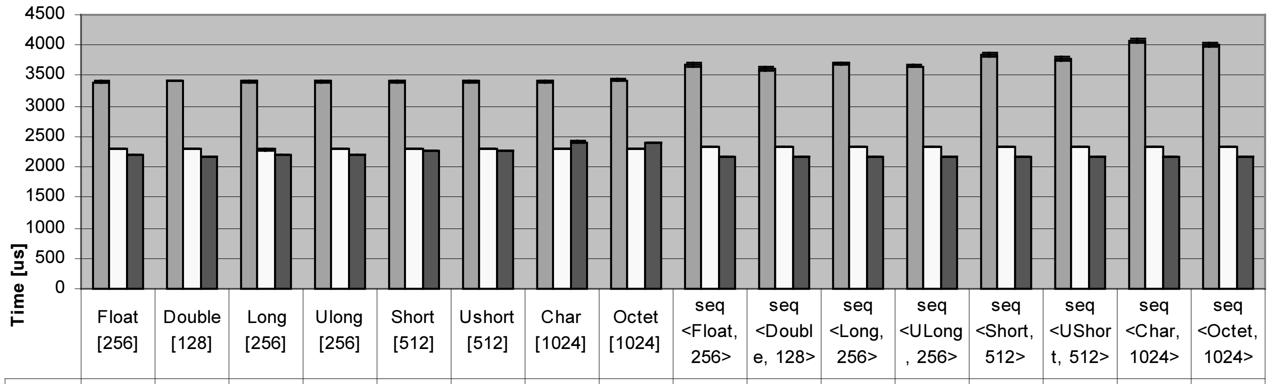
- Of the client-server pair, it is the server who influences the performance most (even when empty operations are called).

- Proprietary optimizations for local invocations are used by VisiBroker (not so with omniORB and Orbix).



	Float [256]	Double [128]	Long [256]	Ulong [256]	Short [512]	Ushort [512]	Char [1024]	Octet [1024]	seq <Float, 256>	seq <Double, 128>	seq <Long, 256>	seq <ULong, 256>	seq <Short, 512>	seq <UShort, 512>	seq <Char, 1024>	seq <Octet, 1024>
▨ Orbix	2973	2953	2957	2967	2997	3010	3077	3073	3070	3037	3097	3067	3233	3170	3483	3420
□ VisiB	1920	1913	1927	1923	1983	1990	2030	2030	1907	1907	1903	1917	1907	1913	1907	1907
■ omni	1806	1749	1810	1816	1928	1939	2115	2109	1726	1723	1722	1726	1728	1728	1725	1731

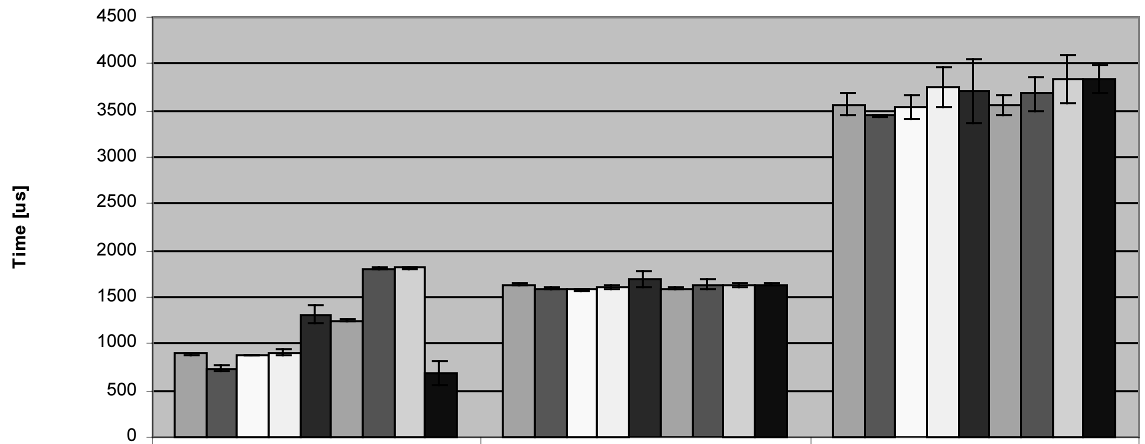
Fig. 49: *THROUGH.IN*: **omniORB 2.5.0** server (**NT** AMD200), ... clients (**NT** Int166), idle net (1 call)



	Float [256]	Double [128]	Long [256]	Ulong [256]	Short [512]	Ushort [512]	Char [1024]	Octet [1024]	seq <Float, 256>	seq <Double, 128>	seq <Long, 256>	seq <ULong, 256>	seq <Short, 512>	seq <UShort, 512>	seq <Char, 1024>	seq <Octet, 1024>
▨ Orbix	3390	3410	3407	3400	3410	3403	3407	3420	3670	3623	3697	3660	3830	3773	4087	4013
□ VisiB	2297	2303	2293	2290	2300	2293	2290	2293	2337	2330	2323	2327	2330	2337	2323	2333
■ omni	2203	2170	2200	2207	2273	2277	2410	2397	2177	2170	2167	2170	2170	2167	2170	2170

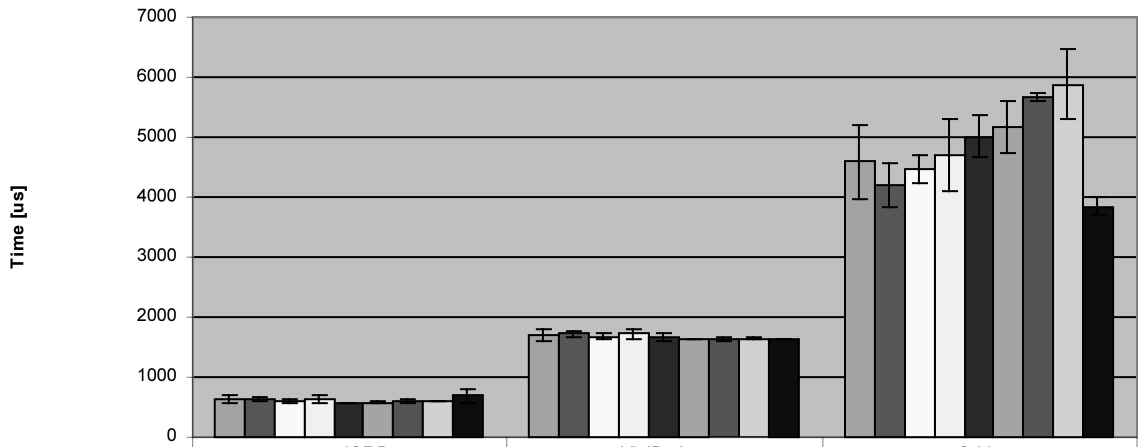
Fig. 50: *THROUGH.IN*: **VisiBroker/C++ 3.0** server (**NT** AMD200), ... clients (**NT** Int166), idle net (1 call)

Because of the interoperability problems (see section 9.1.1.2), some of the measurements with Orbix acting as a server and other ORBs as clients were not done.



	omniORB	VisiBroker	Orbix
▨ Float [256]	890	1633	3567
■ Double [128]	737	1593	3440
□ Long [256]	880	1583	3533
□ Ulong [256]	910	1605	3750
■ Short [512]	1313	1690	3713
▨ Ushort [512]	1253	1593	3560
■ Char [1024]	1807	1637	3677
□ Octet [1024]	1813	1620	3837
■ String <1024>	687	1633	3840

Fig. 51: *THROUGH.IN*: homogenous combinations on **AIX** (passing arrays—1 call)



	omniORB	VisiBroker	Orbix
▨ seq <Float, 256>	640	1697	4587
■ seq <Double, 128>	627	1723	4203
□ seq <Long, 256>	607	1677	4463
□ seq <ULong, 256>	640	1720	4703
■ seq <Short, 512>	570	1673	5010
▨ seq <UShort, 512>	577	1627	5160
■ seq <Char, 1024>	610	1643	5653
□ seq <Octet, 1024>	600	1643	5880
■ String <1024>	687	1633	3840

Fig. 52: *THROUGH.IN:* homogenous combinations on **AIX** (passing sequences—1call)

9.1.1.2 Interoperability

The following table describes results of the basic interoperability tests (running the THROUGH.IN benchmark with different combinations of clients and servers):

Server → / **Client ↓**	NT Omn25	NT Vis30	NT Orb22	NT Orb23	AIX Omn25	AIX Vis30	AIX Orb22
NT Omn25	OK	OK	client hangs	client aborts	OK	OK	sequences[*]
NT Vis30	OK	OK	client hangs	client aborts	OK	OK	sequences[*]
NT Orb22	OK	OK	OK	client fakes[+]	OK	OK	sequences[*]
NT Orb23	narrow[$]	narrow[$]	OK	OK	narrow[$]	narrow[$]	sequences[*]
AIX Omn25	OK	OK	client hangs	client aborts	OK	OK	sequences[*]
AIX Vis30	OK	OK	client hangs	client aborts	OK	OK	sequences[*]
AIX Orb22	seq of any[#]	OK	OK	client fakes[+]	seq of any[#]	OK	OK

The interoperability problems can be summarized as follows:

- A server running Orbix on Windows NT cannot be accessed from other than Orbix clients (both NT and AIX clients work, only an Orbix 2.2 client cannot access the Orbix 2.3 server). Most likely, this is an interoperability problem with Orbix (see section 7.1.1.5).
- Orbix 2.2MT on AIX 4.1 has problems passing sequences outside modules. No client except AIX Orbix was able to pass a sequence to the AIX Orbix server. The AIX Orbix client was not able to pass a sequence to the omniORB 2.5 server (both NT and AIX servers).

No other problems were encountered regarding interoperability of the individual brokers.

[*] With modules OK, without modules client aborts when trying to pass a sequence.

[+] Client acts as if it is connected to the server but in fact no communication takes place.

[$] Without modules OK, with modules client fails to narrow the object reference.

[#] With modules OK, without modules client aborts when trying to pass a sequence of anys.

9.1.2 Nonstandard Extensions

Extension	omniORB	Orbix	VisiBroker
Locating Objects	no (except Naming Service)	Locators (based on service name)	Location Service (based on interface name)
Binding To Objects	no (except standard calls)	_bind (location service, deferred, rebind, timeouts)	_bind (location service, deferred, rebind, timeouts)
Instantiating Implementations	`obj_is_ready`, `dispose`	implicit, `release`, `dispose`	`obj_is_ready`, `release`, `deactivate_obj`
Persistent References	no (except object key)	yes (Loaders)	yes (Activators)
Automatic Activation	no	yes	yes
Automatic Reconnection	no	yes	yes (but masks even important errors)
Invocation Extensions	custom proxies	custom proxies	custom proxies, optimized local calls
Customizability	no (but source is available)	yes (Filters)	yes (Interceptors)

9.1.2.1 Multi-threading Extensions

All three ORBs support multithreading on both the server and the client sides. The Orbix ORB is most flexible with respect to employing different threading strategies, with omniORB and VisiBroker offering practically no flexibility (VisiBroker with its three built-in threading strategies is slightly better than omniORB with one). For details, see the corresponding sections in the individual ORB chapters.

The cross-platform execution of the MT.CONN benchmark demonstrates a limitation of the omniORB server threading strategy. The omniORB server is unable to process requests sent in parallel over a single connection, thus it serializes requests from multi-threaded clients that send requests over a single connection:

Server →	omniORB	Orbix	VisiBroker
Client ↓			
omniORB	50s	N/A	50s
Orbix	250s	50s	50s
VisiBroker	250s	N/A	50s

9.1.2.2 Management Extensions

Both Orbix and VisiBroker provide ORB management support, see the corresponding sections within the individual ORB chapters for details. No management support is provided with omniORB.

9.1.3 Scalability

9.1.3.1 Speed vs. Number of Objects

9.1.3.1.1 PROXY

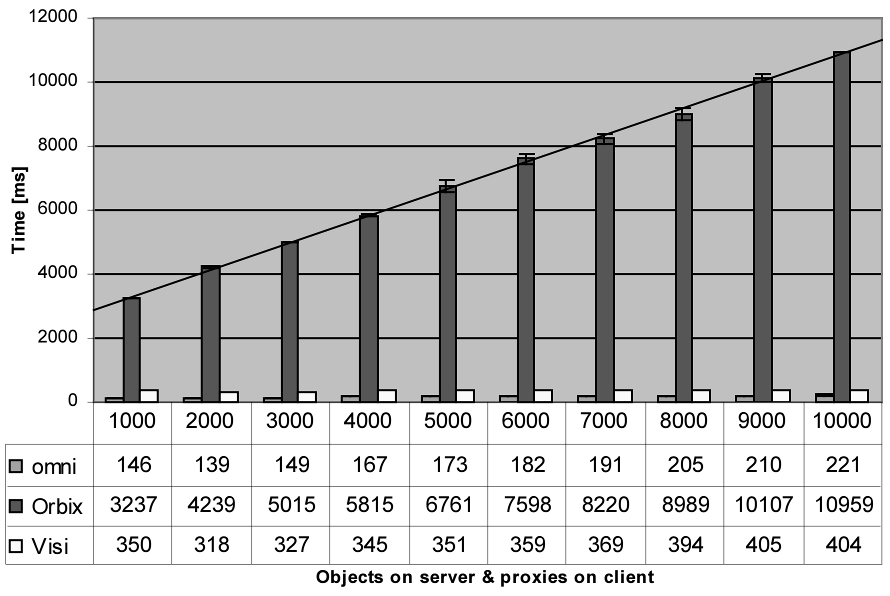
The PROXY.CREATE benchmark indicates dependence between the number of objects and the object creation time both on the server and the client sides (except for VisiBroker). The PROXY.FIRST and PROXY.NEXT benchmarks show that:

- There is a significant difference between the time it takes to invoke an operation for the first time and for all subsequent times.

9.1.3.1.2 PROXY.LOT

The PROXY.LOT benchmark shows that:

- The time to create and register an object on the server side grows linearly with the number of objects with omniORB and Orbix (no difference with VisiBroker).
- The time to create an object proxy on the client side grows linearly with the number of objects with Orbix (no difference with omniORB and VisiBroker).
- When present, the difference in object creation time is significant for a medium number of objects (thousands).
- The time it takes to invoke an operation grows linearly with the number of objects on the server , but compared to the total invocation time, the growth is relatively insignificant.



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▨ omni	146	139	149	167	173	182	191	205	210	221
■ Orbix	3237	4239	5015	5815	6761	7598	8220	8989	10107	10959
□ Visi	350	318	327	345	351	359	369	394	405	404

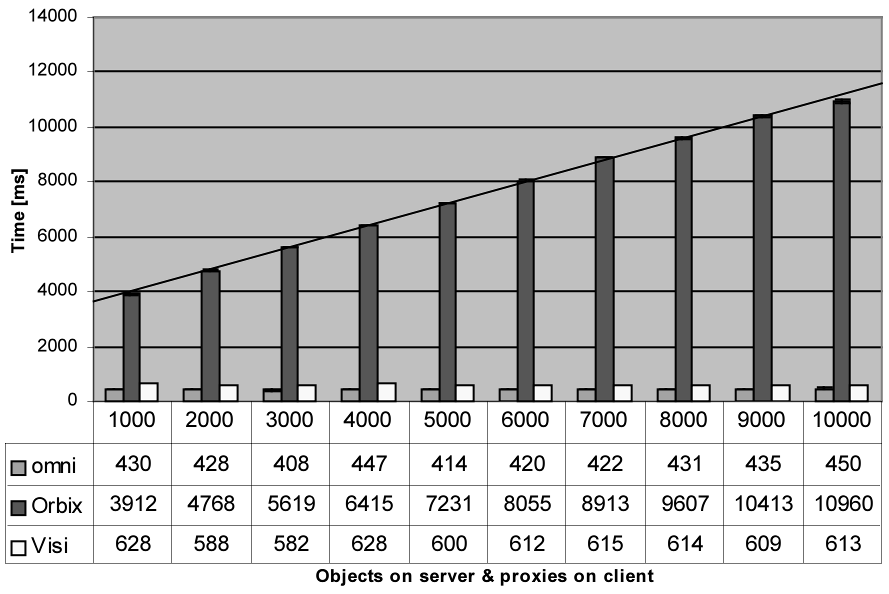
Fig. 53: *PROXY.LOT*: **omniORB** 2.5.0 server NT AMD166 local, Get() operation

9.1.3.2 Resources vs. Number of Objects

Memory consumption	omniORB		Orbix		VisiBroker	
	Client	*Server*	*Client*	*Server*	*Client*	*Server*
Per object (in bytes)	313	238	1232	1141	1075	1214

Fig. 54: Table shows the memory consumption per one object

The table shows the memory consumption per single object (average for 10000 objects that implement an interface with a single `void x()` operation and no attributes).



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
▒ omni	430	428	408	447	414	420	422	431	435	450
■ Orbix	3912	4768	5619	6415	7231	8055	8913	9607	10413	10960
□ Visi	628	588	582	628	600	612	615	614	609	613

Fig. 55: *PROXY.LOT*: **VisiBroker** 3.0 server NT AMD200 local, Get() operation

9.1.3.3 Resources vs. Queued requests

9.1.3.3.1 NET.ONEWAY

The results of the NET.ONEWAY benchmark show that with all ORBs, oneway calls can either block the client (this usually happens when around 300-500 requests are waiting to be serviced and is probably related to the size of communication buffers along the request delivery path), or exhaust the server resources (most often, virtual memory is consumed and the server crashes).

9.1.4 Robustness

9.1.4.1 Reliability

The three tested ORBs all performed reliably, except for an occasional crash of the omniORB server running on AIX (which might have been caused by an incorrect AIX port). The support for building reliable servers is compared in the following table:

Reliability	**omniORB**	**Orbix**	**VisiBroker**
Reliable Servers	persistent references, migration (limited)	dynamic activation, rebinding	dynamic activation, rebinding, migration
Network Failures	nothing	rebinding	rebinding, migration
Network Congestion	migration	load balancing on bind (limited)	load balancing on bind (limited)

9.2 Services

Services	**omniORB**	**Orbix**	**VisiBroker**
Naming	**Yes** (persistent bindings)	**Yes** (persistent bindings, group bindings)	**Yes** (persistent bindings)
Lifecycle	**Yes** (move, delete)	**No**	**No**
Event	**No**	**Yes** (untyped and typed push, untyped reliable persistent push and pull)	**Yes** (untyped push and pull)
Trading	**No**	**Yes** (linked trader, persistent offers)	**No**
Security	**No**	**Yes** (SSL, COS Security)	**Yes** (SSL)
Transaction	**No**	**Yes** (OrbixOTS)	**Yes** (TPBroker, VisiBroker ITS)
Database	**No**	**Yes** (ObjectStore, Versant, generic)	**Yes** (VisiChannel for JDBC)

Note that all services but the omniORB Naming are purchased separately.

9.3 Miscellanea

9.3.1 Development Support

The following table lists the support for various useful development features within the individual ORBs:

Feature	**omniORB**	**Orbix**	**VisiBroker**
Access to the reference count value	No	**Yes**	**Yes**
Warn when negative reference count	**Yes**	No	No
Range checking on sequences	**Yes**	No	**Yes**
Generation of skeleton implementation	No	**Yes**	No
Symbolic exception dumping to stream	No	**Yes**	**Yes**
Broker activity logging	**Yes**	**Yes**	No
Reference parsing utilities	**Yes**	No	**Yes**[1]

[1] The reference parsing utility was not a part of the VisiBroker installation, but was sent to us upon request by the VisiBroker Technical Support.

10 Acknowledgements

During the project, Bernhard Beyl and Stefan Tilkov of MLC Systeme GmbH provided valuable comments.

IONA Technologies PLC has kindly provided a free installation of Orbix for the purpose of evaluation.

11 References

Bak97A	Baker, S.: CORBA Distributed Objects: Using Orbix, Addison Wesley, 1997.
Bal96A	Balek, D.: Relationship Service in CORBA, Master Thesis, Charles University, 1996.
Bir84A	Birrell, A. D. & Nelson, B. J.: Implementing Remote Procedure Calls, Xerox Palo Alto Research Center, ACM Transactions on Computer Systems Vol. 2 No. 1 pp 39-59, ACM, 1984.
Car96A	Carando, P.: Toward the Development of an Object Request Broker Evaluation Instrument, presented at OOPSLA '96 Workshop on Large Persistent and Distributed Systems, 1996.
Gok96A	Gokhale, A. S. & Schmidt, D. C.: Measuring the Performance of Communication Middleware on High-speed Networks, in proceedings of SIGCOMM '96, ACM, 1996.
Gok97A	Gokhale, A. S. & Schmidt, D. C.: Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks, IEEE Transactions on Computers, 1997.
Har97A	Harrison, T. H. & Levine, D. L. & Schmidt, D. C.: The Design and Performance of a Real-time CORBA Event Service, in proceedings of OOPSLA'97, ACM, 1997.
Hri96A	Hrinko, S.: LifeCycle Service in CORBA, Master Thesis, Charles University, 1996.
Ion95A	Orbix 2 Distributed Object Technology Programming Guide Release 2.0, IONA Technologies, 1995.
Ion95B	Orbix 2 Distributed Object Technology Reference Guide Release 2.0, IONA Technologies, 1995.
Kle96A	Kleindienst, J. & Plasil, F. & Tuma, P.: CORBA and Object Services, in proceedings of SOFSEM '96 pp 74-93, Springer-Verlag LNCS 1175, 1996.
Kle96B	Kleindienst, J. & Plasil, F. & Tuma, P.: Lessons Learned from Implementing the CORBA Persistent Object Service, in proceedings of OOPSLA '96, ACM, 1996.
Kle96C	Kleindienst, J. & Plasil, F. & Tuma, P.: What We Are Missing in the CORBA Persistent Object Service Specification, presented at OOPSLA '96 Workshop on Large Persistent and Distributed Systems, 1996.
Lam91A	Lamb, C. & Landis, G. & Orenstein, J. & Weinreb, D.: The ObjectStore database system, CACM Vol. 34 No. 10 pp 50-63, ACM, 1991.
OMG92A	Object Services Request For Proposal 1, OMG document 92-8-6, ftp://ftp.omg.org, 1992.
OMG92B	Object Services Architecture Revision 8.1, OMG document 95-1-47, ftp://ftp.omg.org, 1992-1995.
OMG93A	CORBAservices Specification, Event Management Service, OMG document FORMAL/97-2-9, ftp://ftp.omg.org, 1993-1997.

OMG93B CORBAservices Specification, Lifecycle Service, OMG document FORMAL/97-2-11, ftp://ftp.omg.org, 1993-1997.

OMG93C CORBAservices Specification, Naming Service, OMG document FORMAL/97-2-8, ftp://ftp.omg.org, 1993-1997.

OMG94A CORBAservices Specification, Object Transaction Service, OMG document ORBOS/97-3-4, ftp://ftp.omg.org, 1994-1997.

OMG94B CORBAservices Specification, Relationship Service, OMG document FORMAL/97-2-14, ftp://ftp.omg.org, 1994-1997.

OMG94C CORBAservices Specification, Object Externalization Service, OMG document FORMAL/97-2-13, ftp://ftp.omg.org, 1994-1997.

OMG94D CORBAservices Specification, Persistent Object Service Version 1.0, OMG document FORMAL/97-2-10, ftp://ftp.omg.org, 1994-1997.

OMG94E CORBAservices Specification, Concurrency Service, OMG document FORMAL/97-2-12, ftp://ftp.omg.org, 1994-1997.

OMG95A ORB Portability Enhancement RFP, OMG document 95-6-26, ftp://ftp.omg.org, 1995.

OMG96A Common Object Request Broker Architecture And Specification Revision 2.0, OMG document PTC/96-3-4, ftp://ftp.omg.org, 1996.

OMG96D CORBAservices Specification, Security Service Part I, OMG document FORMAL/97-2-20, ftp://ftp.omg.org, 1996-1997.

OMG96E CORBAservices Specification, Security Service Part II, OMG document FORMAL/97-2-21, ftp://ftp.omg.org, 1996-1997.

OMG96G CORBAservices Specification, Trader Service, OMG document FORMAL/97-5-1, ftp://ftp.omg.org, 1996-1997.

OMG97C Persistent State Service Version 2.0 Request For Proposal, OMG document ORBOS/97-6-7, ftp://ftp.omg.org, 1997.

OMG97D ORB Portability Joint Submission Part 1 of 2, OMG document ORBOS/97-5-15, ftp://ftp.omg.org, 1997.

OMG97E ORB Portability Joint Submission Part 2 of 2, OMG document ORBOS/97-5-16, ftp://ftp.omg.org, 1997.

OMG97J CORBAservices Specification, Full Book, OMG document FORMAL/97-6-1, ftp://ftp.omg.org, 1997.

OMG97M CORBAservices Specification, Trader Update Package, OMG document FORMAL/97-6-18, ftp://ftp.omg.org, 1997.

OMG97N The Common Object Request Broker: Architecture and Specification Revision 2.2, OMG document FORMAL/98-2-1, ftp://ftp.omg.org, 1998.

OMG97N A Discussion of the Object Management Architecture, The Object Management Group, http://www.omg.org, January 1997.

ORL97A Interoperability Between omniORB2 and Java ORBs, ORL, http://www.orl.co.uk/omniORB/javaORBs.html, 1997.

Sha86A Shapiro, M.: Structure and encapsulation in distributed systems: the proxy principle, in proceedings of The 6th International Conference on Distributed Computer Systems pp 198-204, Boston, 1986.

Tum97A Tuma, P.: Persistence in CORBA, Ph.D. Thesis, Charles University, 1997.