Beeb.

Comparative Performance Evaluation of CORBA and DCOM

R.L.J. Beekhuis

Informatica

RuG

Afstudeerverslag

Comparative Performance Evaluation of CORBA and DCOM

R.L.J. Beekhuis

Begeleiders:
Prof.dr.ir. L.J.M. Nieuwenhuis
Ir. A.T. van Halteren
Rijksuniversiteit Groningen
Informatica
Postbus 800
9700 AV Groningen

april 1998

Author: R.L.J.Beekhuis

Date: April 1998

Comparative Performance Evaluation of CORBA and DCOM

methodology, measurements and results

For internal use only at department of Computing Science
at University Groningen

Contents

Abstract

Developments in the area of computing systems and computer networks contribute to an increasing field of applications for distributed systems. In practice, the environment for distributed applications is heterogeneous, i.e., software components are developed in different programming languages, and executed by computers from various vendors running different types of operating systems. Distributed application platforms provide a means to reduce the complexity of designing and operating distributed applications in such heterogeneous environments. A distributed application platform hides the heterogeneity of the underlying computer network. A distributed application platform provides the application programmer with a set of uniform, standardised generic functions, which support the application. The distributed application platform is built using middleware, i.e., a range of software products to support various interactions between components of the distributed application. Object Request Brokers (ORB's) are an example of such middleware products. The ORB developments are dominated by CORBA and DCOM. CORBA is OMG's Common Object Request Broker Architecture. A number of software vendors have developed CORBA-compliant ORB's. DCOM is developed by Microsoft and used in a number of their products.

In this report, I present the results of my graduation assignment, which has been carried out at KPN Research in Groningen. ORB's are relatively new, and little is known about the performance of ORB's when they are applied in large scale applications. Furthermore, there is a strong competition between CORBA and DCOM. Both aspects justify a study on the performance of Object Request Brokers.

The literature study made clear that performance evaluation of such complex systems is far from trivial. I therefore first studied the architecture of CORBA and DCOM. These analysis resulted in a generic model for ORB's. As a next step, from the various performance evaluation strategies I selected a measurement strategy based on software implemented monitoring. A number of external factors, e.g., network latency, that could disturb the measurements have been identified and taken into account. The test suite based on carefully constructed benchmarks has been defined. Then, a test programme comprising twelve experiments has been developed. A subset of the experiments where implemented to evaluate Microsoft's DCOM and the CORBA 2.0 implementation Orbix of Iona Technologies. The benchmarks that where executed on DCOM and CORBA, are especially intended to evaluate their behaviour on transferring requests with different parameter-types and sizes. Other benchmarks where used to evaluate the behaviour of the middleware products under a sustained workload of a large number of clients doing requests.

List of figures

List of Abbreviations

APPID	Application Identifier
BOA	Basic Object Adapter
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CSMA/CD	Carrier Sense, Multiple Access with Collision Detection
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
CLSID	Class Identifier
DII	Dynamic Interface Invocation
DSI	Dynamic Skeleton Invocation
GUID	Globally Unique Identifier
IDL	Interface Definition Language
IIOP	Internet Inter-Orb Protocol
IPID	Interface Pointer Identifier
IT	Information Technology
MAC	Media Access Control
MIDL	Microsoft Interface Definition Language
NIC	Network Interface Card
OA	Object Adapter
ODL	Object Definition Language
OLE	Object Linking and Embedding
OMG	Object Management Group
ORB	Object Request Broker
ORPC	Object Remote Procedure Call
OSF	Open Software Foundation
OXID	Object Exporter Identifier
RPC	Remote Procedure Call
SCM	Service Control Manager
SMP	Symmetric Multiple Processor

1 Introduction

This chapter presents the rationale and the objective for the graduation assignment described in this thesis. The succeeding sections give the reader an outline of this thesis.

1.1 Assignment

This thesis presents a methodology for measuring the performance of middleware. This methodology is used for the performance evaluation of CORBA and DCOM. The author is a graduate student of the faculty of Computer Science at the University of Groningen. This report serves as the final deliverable for a graduate assignment. In this particular case, the assignment was done at the department CAS (Communication Architectures & open Systems) of KPN Research at their site in Groningen.

1.2 Rationale

As information technology becomes an integrated part of our modern society, applications are being developed that can't reside on one computer-system anymore. Reasons for this phenomenon can be the number of resources an application needs or the geographical distribution of certain parts of an application. This division of applications in separate parts and distributing them to run on separate computer-systems is generally referred to as 'distributed computing'. Because 'distributed computing' needs more than just a network, several standards were defined to provide a software infra-structure with well defined interfaces for application developers.

Although the fact that the mentioned software infrastructures provide a standard for the communication between parts of an application, the heterogeneous use of hardware and operating systems commonly introduced interoperability problems. Besides the interoperability problems, there was also no standardisation on the services such a software infrastructure should provide and the implementation of these services.

To overcome these problems, distributed computing architectures have been defined. These architectures specify the software infrastructure and additional services needed for distributed computing. The implementations of these software infrastructures and the additional services are referred to as middleware. Two leading architectures for middleware are CORBA from the Object Management Group (OMG) and DCOM from Microsoft.

Although middleware solves many interoperability issues and widely accepted standards solve many other co-operation problems, the idea that middleware is only advantageous, isn't entirely true. Generally speaking, it can be said that solving interoperability issues will give performance drawbacks through conversions of requests and their parameters. Furthermore the distribution of application components over multiple computer-systems might introduce excessive network latency on requests between closely interacting components of the application.

Middleware platforms enable designers of distributed systems to fully concentrate on the functionality of the system instead of worrying about interoperability questions. As a result the distributed systems grow larger and become more and more complex. A problem that may arise here, is that there is no common knowledge about the performance of middleware when being applied in large systems. Therefore a methodology for measuring the overhead and scale-ability behaviour of these platforms is a desirable tool for evaluating implementations of commercially available products.

1.3 Objective

The graduate assignment includes the development of 'a methodology for performance and scale-ability measurements on middleware'. To get a clear view on the architecture of this middleware, this thesis provides an architectural description of two middleware products. Based on this evaluation, a generic architecture for middleware infrastructures will be composed. This generic architecture will be used as a fundament for the development of a methodology for evaluating the performance and will result in the description of several benchmarks. Some of these benchmarks are implemented and applied on the CORBA implementation Orbix and on Microsoft's DCOM, to evaluate their performance and the effectiveness of the benchmarks.

1.4 Thesis structure

To enable easier reading, this section gives a brief overview of the content of the upcoming chapters. The chapter-descriptions give an overview of the structure of the chapters and topics discussed in that particular chapter.

- **Chapter 1** gives an introduction to this thesis, by giving the rationale and objectives described in this thesis.
- **Chapter 2** describes middleware in general. From this general perspective, two leading middleware infrastructures are functionally decomposed. This decomposition is used to create a generic middleware architecture, that is used in the subsequent chapters.
- **Chapter 3** provides information about performance evaluation and the techniques that are frequently used. From all these techniques and methods, the most suitable method is chosen to be used for commercial middleware products.
- **Chapter 4** describes a collection of experiments, which form the basis of our benchmark-suite. Some of these benchmarks were developed and executed, to evaluate the performance of the middleware infrastructures DCOM and a CORBA implementation. The results of these benchmarks and their specific implementation details are also given.
- **Chapter 5** gives the conclusions about the methodology and the results of the evaluated infrastructures, and recommendations for future work and research.

2 Middleware architectures

In many cases, a middleware infrastructure is seen as the glue that connects objects spread over multiple heterogeneous computer-systems. This viewpoint shows little or nothing of the internal organisation and architecture of the specific middleware infrastructure.

The objective of this chapter therefore, is to find a generic architecture of middleware infrastructure. To get a clear view on middleware and their services, a general model will be discussed first. This shows the position of middleware in a distributed system. From this general overview, two leading middleware architectures (CORBA and DCOM) are evaluated in the subsequent sections. The evaluation will start with a brief overview of the history of the middleware product. After this small introduction, the architecture is evaluated and the functionality of the parts is described. The evaluation is concluded with an overview of the services and a short description about upcoming and future enhancements in the architecture and services.

Based on the analysis of CORBA and DCOM, a generic architecture for middleware is described.

2.1 Generic concepts

Before describing middleware in general or evaluating the two example architectures, some knowledge about the data-conversions that occur for transport is necessary. This converting from requests and their parameters is called marshalling and the de-conversion unmarshalling.

2.1.1 (Un)marshalling

Because of the heterogeneity of the hardware, operating systems and the supported programming languages, a common representation of the transported data must be used. This means that the network-layer in the computer-system must always present the exact byte stream to the middleware, independent of environmental differences.

Marshalling means that all the request information in copied into a flattened buffer of bytes. This process may sound trivial, but in fact it is not that trivial at all. While the data that must be marshalled can be a simple integer, it can also be a structure with pointer references to lists and other structures. This means that the middleware component responsible for the marshalling process, must traverse all these pointers and structures, and copy their contents in the buffer.

Unmarshalling is the reverse procedure in which the data in the flattened buffer must be rebuild to its original form. This means that the unmarshalling code is responsible for setting up the environment for a local method invocation or procedure call.

The unmarshalling code performs the following actions:

- shuffling data-bytes to the native byte order (big- or little-endian). The sender transmits its data in its own native byte order and passes a flag in the header of the request to indicate its byte order. A receiver checks this flag, and depending on its own native byte order, it must re-shuffle the data-bytes.
- rebuilding the data-structures. To obtain the information that was available on the sender-side, the raw-data must be converted to the same data types. This means that in the case of complex data types, the unmarshalling code must reconstruct the data-structure, or could even be required to rebuild complete lists.

- rebuilding the method invocation/procedure call. Before the actual call is invoked on the server-object or the right procedure is called, the unmarshalling code reconstructs the complete callers environment. This means that the complete calling-stack must be rebuild from the data it just reconstructed in the previous phase.

2.1.2 Middleware functions

A middleware-infrastructure can be seen as an extension of the operating system, which provides a transparent communication layer to the applications using the middleware.
In a picture it looks like this:

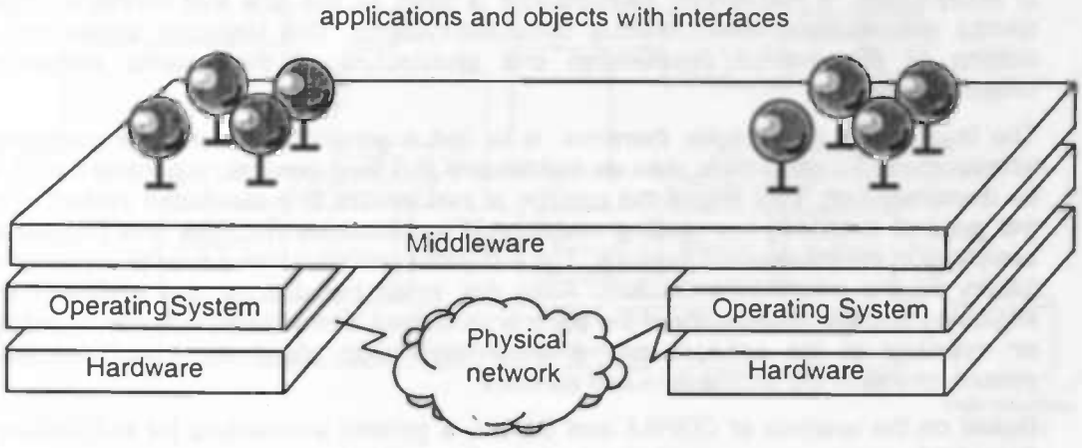


Figure 1:Position of middleware

As can be seen in Figure 1, the middleware is positioned on top of the hardware and the operating system, and underneath the applications that use the middleware to communicate across machine boundaries. It is as though the middleware provides a virtual machine to the applications. One of the goals of middleware is, to supply interoperability between different heterogeneous hardware and operating systems. For the application-objects it shouldn't matter where they are located, they can always communicate with another.

Because of the interoperability of the heterogeneous systems, the middleware not only provides the means to communicate, but mostly offers one or more of the following services and transparencies:

- interaction transparency: In order to let different heterogeneous hardware platforms transport data, like function/method parameters, a common data representation is used by the middleware. This means that for every environment a middleware product supports, this same representation is used for all the data-primitives that can be used in transport.
- location transparency: If a client wants to perform some sort of communication with a server-object, a request is issued to the middleware-platform. The middleware locates the server-object by querying a database for the location of the requested object and invokes the server-object. This means that for the client it doesn't matter where the server-object is located.
- garbage-collection: When a server-object is no longer needed by a client, because the client has finished the communication or by a broken connection, the server-object is brought down to save resources. Typical means are time-outs, reference counts and pinging methods.
- security / authentication: By using a distributed environment, where a lot of people can access a server-object, a means of security must be provided so confidential information remains confidential. User-names and passwords gives a means of authentication, and encryption can be used for the security of the data.

- migration transparency: Besides the location-transparency, server-objects can be migrated from one computer-system to another. With migration, the objects are not only launched on another computer-system, but the internal state of the object being migrated is transferred from its originator to the newly created object.
- load-balancing: Load-balancing is typically used when there are too many servers-objects residing on a computer-system. When a system has too many server-objects, migration of one or more server-objects to a computer-system with less load can improve the performance.
- fault-tolerance: Distributed systems are usually connected by means of a network, which may break down. Beside these failures, the computer-systems themselves can also fail, which can cause a server-object to fail or be unreachable. Providing fault-tolerance gives a higher degree of availability by using redundant objects, or by using multiple objects that perform the exact same functions and have the same internal state. This kind of fault-tolerance is called replication.

2.2 CORBA

2.2.1 History

CORBA is a specification for middleware, defined by the Object Management Group (OMG) founded in May 1989. This is a standardisation group which consists of over more then 700 members. Those members include the biggest IT-vendors in the world, i.e. SUN, Oracle, Unisys and IBM. The goal of the OMG was to come to a specification how a middleware platform should behave and what services it should provide.

In the first specification of CORBA, the protocol on how Object Request Brokers (ORB's) should communicate with each other was not specified. This led to problems when ORB's from different vendors had to communicate / exchange data. This and several reasons made a new version of the CORBA specification necessary. The new specification led to CORBA 2.0, with a mandatory requirement for IIOP, which stands for Internet Inter-ORB Protocol. This specification describes the protocol that ORB's should use for their communication, so that ORB's from different vendors can be used. This feature is of great importance when CORBA should be used on different run-time environments, where ORB implementations from different vendors are used.

CORBA therefore is not a product, only a specification with a well defined interface. Because the specification is free, numerous different implementations of this specification exist. Implementations can be found from IT-vendors with commercial products but also freeware versions. Of course the functionality of these products differ a lot. Differences can be found in programming language support, the number of services that are provided and the number of hardware-platforms the product supports.

2.2.2 Generic CORBA concepts

Before diving into the architecture of CORBA, some generic concepts used throughout the whole architecture need to be described. A little knowledge of these concepts is essential for a good understanding of the functionality of the components in the architecture.

Interface Definition Language (IDL)

Interfaces in CORBA are described in the Interface Definition Language (IDL) which is specified by the OMG. IDL is a descriptive language, in which the interfaces a server-object exposes are defined. When clients and server-objects interact, the interfaces and their methods are identified by the name they get in IDL. When the interfaces are defined, the IDL compiler (provided with the development environment for a specific CORBA implementation) must be used to generate the marshalling and unmarshalling code for the client and server-object.

Common Data Representation (CDR)

The Common Data Representation (CDR), is OMG's specification for transporting requests and their parameters in CORBA. The specification consist of a description how certain primitive data-types, and complex structures based on these data-types must be copied into a flat memory-buffer, which is the actual marshalling process.

2.2.3 Architecture

The following picture shows the decomposition of the CORBA-ORB architecture [OMG-97][ORFALI-97B].

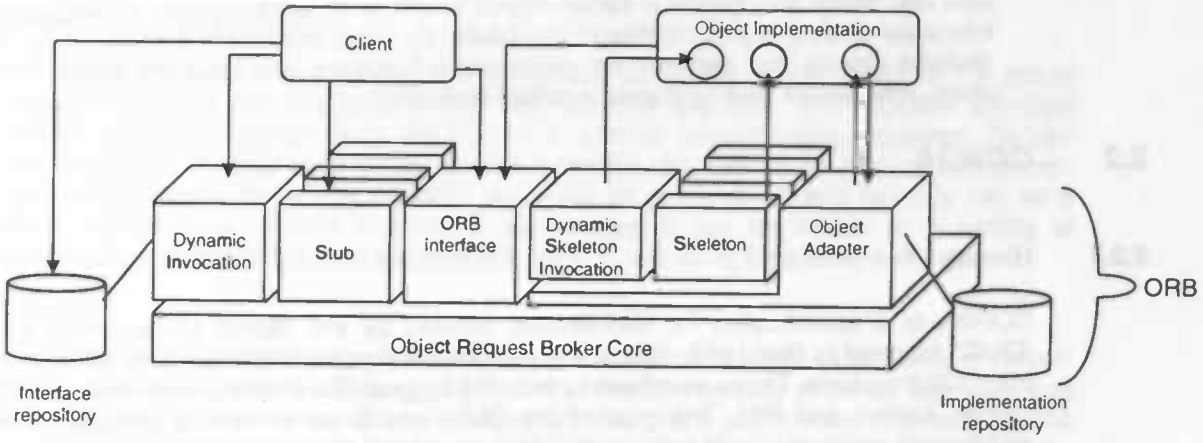


Figure 2:CORBA architecture

Âs already mentioned, CORBA is a specification, whereby the implementation is free to the vendor of a specific product. This means that a component of the middleware platform can only be described by the functionality it should provide, without any information about its specific internals.

The following sections, therefore give a description about the components of which the CORBA-ORB exists and the functionality they provide.

Object Request Broker Core (ORB-Core)

The ORB-Core is the core-component for the communication between clients and server-objects. It shields the client and server-object form the transport-protocol that is used for their message-exchange.

Where the interfaces between the components of which the ORB exists are specified, and IIOP specifies the protocol between ORB's, the following underlying communication mechanisms can be used:

- RPC (Remote Procedure Calls), for invoking methods on remote objects (client is blocking).
- Send/Receive messages, for non-blocking calls where a response from the server is expected.
- Datagram messages, for send-only messages.

ORB Interface

The ORB Interface is the interface that directly communicates with the ORB core. This component is the same for all ORB's and does not depend on any other component in the architecture. Because most of the functionality is provided through the other components of the architecture, there are only a few operations that are common across all objects. These operations are available to clients and server-objects. Common operations the ORB Interface provides are, converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface (DII) described below .

Stub and Skeleton

Clients and server-objects define an interface specification in IDL(Interface Definition Language). This interface-definition is compiled to a stub on the client-side and a skeleton on the server-side. The stub and skeleton are the connection of the client and the server-object to the ORB-core. The stub is located with the client, and creates and issues request invocations on its behalf. The server-object's skeleton provides a mechanism to deliver the request invocations.

When a client invokes a request on a server-object, the client's stub will be invoked. The stub marshals the request and its parameters from the format used in the client's environment to a representation suitable for transportation on the network. The transport format of requests is specified in the CDR (Common Data Representation).

Once the invocation request arrives at the designated server-object, the ORB-Core and skeleton co-operate to unmarshal the request from its CDR-format, to a programming language form and dispatches this to the right server-object. After the request is completed by the server-object, the skeleton and ORB marshal the response and send it back to the requesting client, where it arrives at the stub.

Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI)

The DII allows a client to access an underlying request mechanism provided by an ORB to dynamically query for the interfaces and methods a server-object supports. Applications use the DII to dynamically issue requests to server-objects without requiring IDL interface-specific stubs to be linked in the process or compile time knowledge of the IDL-specification.

Unlike IDL stubs (which only allow RPC-style two-way and one-way requests), the DII also allows clients to make non-blocking deferred asynchronous calls, which separate send and receive operations.

The DSI is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an server-object or ORB-bridge that has no compile-time knowledge of the type of server-object it is implementing. The client making the request need not be aware that the server-object is using the type-specific IDL skeletons or the dynamic skeletons.

Object Adapter

The Object Adapter serves as the glue between the CORBA server-objects and the OBR-core. In other words, it is an interposed server-object that uses delegation to allow a client to invoke requests on an object even though the client doesn't know the server-object's true interface.

The Object Adapter assists the ORB-core with delivering requests to the server-object and activating the server-object. Essentially, the Object Adapter associates server-objects with the ORB-core. Hence, it defines most of the services from the ORB-core that the server-object can depend on. Object Adapters are involved in the following services:

- registration of server-objects
- generation and interpretation of server-object references
- method invocation
- security of interactions
- server-object activation and deactivation
- mapping object references to server-objects

In CORBA 2.0 a standard Object Adapter was specified, that can be used for almost every server-object, this is the so-called Basic Object Adapter (BOA). But Object Adapters can be specialised to provide support for certain server-object styles (such as Object Database Adapters for persistence, library Object Adapters for non-remote objects, and real-time Object Adapters for applications that require QoS guarantees).

Interface Repository

The Interface Repository is the place where descriptions of the interfaces, extracted from the corresponding IDL's, from the various server-objects are stored. In case of a DII, information from the Interface Repository is used to negotiate and select the appropriate interface for the client to communicate with the server-object.

Implementation Repository

The Implementation Repository is the storage place where server-objects are being registered. By registering a server-object, the Object Adapter knows which server-objects it can handle, and where a binary image is physically stored. The Implementation Repository also keeps the start-up parameters that can or must be used for launching a server-object, and the activation and access policies for the server-objects.

Object Implementation

The Object Implementation, referred in this thesis to as the server-object, is the actual implementation of the functionality that the interface exposes. The CORBA standard defines several mappings from the IDL to a specific programming language. Server-objects can be configured to be started on a client's requests or as a persistent server. Server-objects are commonly provided as separate executables and typically run as a single process. As already mentioned, server-objects can be written in a variety of languages, currently mappings are available for C, C++, Java, Smalltalk, and Ada.

Client

The client is the part that requests the ORB and some of its services to locate a server-object. In the case that a server-object is not active it eventually must be launched and started. After a binding has been established, the client can invoke requests on the server-object without knowing the location of it. The client has also no knowledge of the underlying network and network-protocols that are used to transport its request to the server-object.

2.2.4 Services

The services presented in this section are, the so called CORBA-services. This a set of fifteen services that are published standards of the OMG [OMG-97]. These services can be seen as enhancements to complete and augment the functionality of the ORB.

- The **life cycle service** defines operations for creating, copying, moving and deleting server-objects on the software provided by the ORB.
- The **persistence service** provides a single interface for storing server-objects persistently on a variety of storage server-systems. These include object databases, relational databases and simple flat files.
- The **naming service** allows clients connected to the ORB to locate server-objects by name; it also supports federated naming contexts. The service also allows objects to be bound to existing directory-services. These include ISO's X.500, OSF's DCE Cell Directory Service, SUN's NIS+, Novell's NDS and the Internet's LDAP.
- The **event service** allows clients and server-objects connected to the ORB, to dynamically register and unregister their interest in specific events. The service defines a well-known server-object called the *event channel* that collects and distributes events among objects that are loosely coupled.
- The **concurrency control service** provides a lock manager that can obtain locks on behalf of either transactions or threads.
- The **transaction service** provides a two-phase commit co-ordination among recoverable server-objects using either flat or nested transactions.
- The **relationship service** provides a way to create dynamic associations (or links) between server-objects. It also provides mechanisms for traversing the links that group these server-objects. The service can also be used to enforce referential integrity constraints, track containment relationships and for any type of linkage among server-objects.
- The **externalisation service** provides a standard way for getting data into and out of a server-object using a stream-like mechanism.

- The **query service** provides query operations for clients and server-objects. It's a superset of the standard relational-database query language SQL (Structured Query Language). It is based on the upcoming SQL3 specification and OQL (Object Query Language) specified by the Object Database Management Group's (ODMG).
- The **licensing service** provides operations for metering the use of server-objects to ensure fair compensation for their use. The service supports any model of usage control at any point in a server-object's life-cycle. It supports charging per session, per node, per instance, per instance creation and per site.
- The **properties service** provides operations that let you associate named values (or properties) with any server-object. Using this service, you can dynamically associate properties with a server-object's state, for example a title or a date.
- The **time service** provides interfaces for synchronising time in a distributed client and server-object environment.
- The **security service** provides a complete framework for distributed server-object security. It supports authentication, access control lists, confidentiality and non-repudiation. It also manages the delegation of credentials between server-objects.
- The **trader service** provides a sort of "yellow pages" for server-objects. It allows server-objects to publish their services and bid for jobs.
- The **collection service** provides CORBA interfaces to generically create and manipulate the most common collections.

The services mentioned above provide a robust environment for server-objects in a distributed system, and enables a comfortable way to provide its services.

2.2.5 Future enhancements

CORBA 3.0 is expected to be the next major upgrade of this middleware specification. Next to refinements of the existing technologies and services, new features and services will be added.

One of these new features is messaging. The now existing communication way is the synchronous request-reply mechanism. Using this mechanism a request must be replied by the server-object to let the client continue with its work. With messaging a client does not have to wait for the return of its request, it just sends out his request and checks for a reply later. An asynchronous way of communication between the client and server can be used.

Another feature is interface versioning. With interface versioning it becomes possible to used multiple interfaces on one component, where the new interface may have new implementation code, or additions to older interfaces. By default the newest interface will be used, but an older interface or an interface with different implementation code may also be requested.

To copy objects with the maintenance of their state, CORBA 3.0 will support pass-by-value for objects. This means that the state of an object and all of his ancestors is copied into a flattened data-buffer. The state of the object is then copied as part of the remote invocation. The object and its state are transported in normal IIOP encoding.

These new features and services are only a tip of the iceberg of additions planned for CORBA 3.0. To find out more about new additions to the CORBA specification, a visit to the OMG website is a good place to start. The new additions described here are only meant to show the ongoing evaluation, refinement and enhancement of CORBA.

2.3 DCOM

2.3.1 History

DCOM is a middleware architecture defined and implemented by Microsoft. Microsoft is not a member of the OMG, but is one of the major contributors of the Open Software Foundation (OSF). Microsoft decided to create its own middleware architecture, based on the desktop and backoffice operating systems it developed over the last years. It started

with Microsoft's Object Linking and Embedding (OLE) technology which was introduced in 1990 to provide a cut-and-paste capability for Windows. In 1993 it was subsequently extended to OLE2, which should provide a more general form of communication between Windows applications. With OLE2 it is possible to embed one document type inside another (like in this report PowerPoint slides in a Word document). Furthermore OLE2 also provided drag-and-drop capabilities, to allow document components to be picked up from one window and positioned into another. Next to these, OLE automation provided a way to use OLE2-aware applications from scripts, or to co-ordinate applications to perform complex tasks.

With OLE2 a simple single-machine communication model was provided, that was given a separate name: COM, which was said to stand for Component Object Model. Components in C++, Visual Basic and other languages that drove applications by OLE automation came to be called OLE Controls (OCX's), and COM was promoted to be a general purpose infrastructure to create component-based applications.

Microsoft has now provided a set of libraries with the same interface as COM, that enable communication between components on networked computers in the same way as on a single machine. This is called Distributed COM (DCOM), and is incorporated in Windows NT 4.0 and is also available for Windows 95. Third party vendors like DEC, Software AG and Bristol Technologies are working to port DCOM to other platforms, like UNIX and VMS.

2.3.2 Generic DCOM concepts

Before diving into the architecture of DCOM, some generic concepts used throughout the whole architecture need to be described. A little knowledge of these concepts is essential for a good understanding of the functionality of the components in the architecture.

Globally Unique Identifiers

GUID's or **G**lobally **U**nique **ID**entifiers play an important role in (D)COM, as they are identifiers for every part in the whole platform and the applications build on it. GUID's are basically 128 bits numbers, that are really unique. These numbers can be created by a special utility that creates a composite number of the MAC-address (which is a unique 48 bits number) of the machine's NIC, the date and timer and an additional high resolution timer. According to Microsoft, this should guarantee that no duplicate GUID's will be created until somewhere after 5770 AD.

All the interfaces of COM and DCOM servers are identified though these numbers, where they are called IID's (Interface IDentifiers) When an object exposes more than one interface, the object is identified through a CLSID (CLass IDentifier). As a server or application can have multiple objects that expose their interface in a COM-manner, an APPID (APPlication IDentifier) can be used to group multiple CLSID's for the objects that reside in one application. GUID's are not pleasant things to work with, so gladly after the declaration in the interface definition, human readable names can be used through the rest of the application development process.

Microsoft Interface Definition Language (MIDL)

Interfaces in DCOM are described in the Microsoft Interface Definition Language (MIDL). MIDL is a descriptive language, in which the interfaces a server-object exposes are defined. As already mentioned, the interfaces are identified by a GUID. When the interfaces are defined, the MIDL compiler (provided with the development tools that support DCOM) must be used to generate the proxy for a client and the stub for a server-object.

Network Data Representation (NDR)

The Network Data Representation (NDR), is Microsoft's specification for transporting requests and their parameters in DCOM. The specification consist of a description how certain primitive data-types, and complex structures based on these data-types must be copied into a flat memory-buffer, which is the actual marshalling process. The

specification is closely related to the specification used in DCE-RPC, but some additions have been made by Microsoft for the transport of data necessary for DCOM.

Pinging

The technique of pinging is a very common way to provide a keep-alive mechanism. Pinging means that a client sends a message to the server in what it says: "Hey, I'm still alive and like to use your services in the near future". In DCOM the time-frame in which a client should ping its server is default 2 minutes. If a server doesn't get pings from its clients for 3 pinging periods, it may unload itself although there may be still open connections.

When a client has a large number of references to server-objects, Delta-Pinging can be used. Using Delta-Pinging, the client defines a set (a so called ping-set) of object-references. Then the client informs the server, what server-object references are contained in the ping-set, and gives the ID of this ping-set. For the rest of the life-cycle of the clients, only one message is sent that contains the ID of the ping-set. When clients are no longer interested in a certain server-object reference or creates a new server-object reference, it can alter the ping-set. The additions or subtractions to the ping-set are sent to the server on the next ping-message.

2.3.3 Architecture

Figure 3 gives an overview of the DCOM architecture, which is based on [BROWN-98][HORST-97] with some adaptations to the architecture based on various literature[GRIMES-97][EDDON-98].

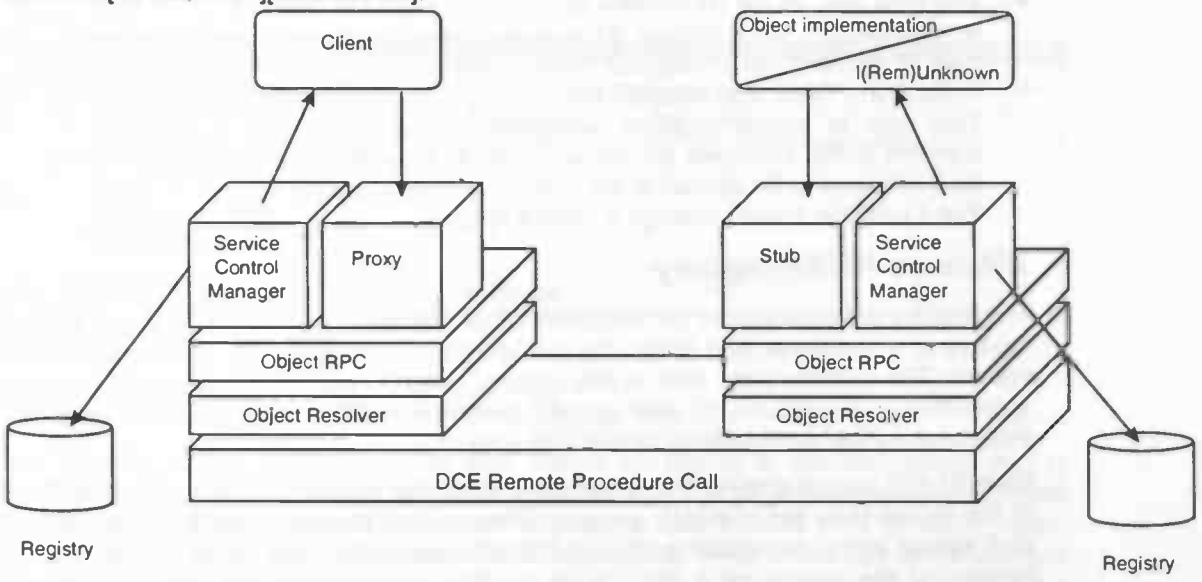


Figure 3:DCOM architecture

In the following sections the DCOM components and their functionality is described.

DCE Remote Procedure Call

The core of DCOM is based on the DCE Remote Procedure Call specified by the OSF. DCE stands for Distributed Computing Environment specified by the Open Software Foundation [BROCKSCHMIDT-95], which provides a complete runtime and development environment for distributed applications. From this specification, Microsoft has integrated the Remote Procedure Call part into their DCOM specification. The RPC mechanism provides cross machine- or process-boundary procedure calls, that are transparent to the programmer.

Object RPC

Object RPC is a small layer above standard RPC, to provide cross process method-invocation between objects. ORPC is a wrapper around DCE RPC that adds additional

mechanisms for DCOM specific functions. These are required to make calls on remote objects, and include functions for object-reference representation, maintenance and transport. Furthermore, it defines the way error messages are returned and what kind of standard DCE RPC functions may or may not be used.

Object implementation

The object implementation, in this context referred to as a server component or server object, is the provider of a piece of functionality a client can use. In DCOM several types of server components exist:

- The first type, is the in-process server. This server is provided as an DLL (Dynamic Link Library) and is loaded in the client's process space when the client requests a binding to the server.
- The second type, is the out-of-process server. This server can be provided as a DLL or an executable.
    - An executable server is loaded as a separate process, just like any other ordinary application.
    - A DLL is meant to be loaded in the process space of the process that binds to the DLL. In this case this process is not available and another solution must be found. The solution is called a *surrogate*. Windows NT4.0 SP2 and DCOM for Windows 95 provide a default surrogate process, as well as a protocol for writing custom surrogates. The default surrogate is an executable server skeleton that is especially designed to load any DCOM DLL-server in its process space. The COM-libraries themselves provide the functionality of starting a surrogate, loading the required DLL and the reverse of this process when the DLL-server is no longer required.
- The third type, is the NT-service server. This server is provided as an executable and is controlled by the Windows NT system. The start-up and stopping behaviour of this type of server must configured through the Service Control Manager applet (in this context the SCM that controls the starting and stopping of Windows NT Services). This type of server requires additional programming over a standard server to conform to the Windows NT Service model. An advantage of this type is, that it can be configured to be started at the start-up of the computer-system. A disadvantage is, that it can't be started through a client's request.

Windows NT/95 Registry

The registry is a core part of the Microsoft 32 bit operating systems NT 4.0 and W'95. The registry is a database that holds the configuration information for a specific computer-system. The configuration data in the registry ranges from hardware configurations and application configurations to user specific preferences. The registry is also used as the place to store the configuration of (D)COM objects.

Each CLSID has an entry in the registry that holds the location of the server and the type of the server (see the previous section). These CLSID entries reside on both the client and server computer-systems. On the server computer, the SCM knows the exact location of the server, so it can be loaded into memory. On the client computer, the location of the server can be local or the name of a remote computer.

Besides the location of an object, a separate registry-key has an IID entry for objects that are used in the computer-system. This entry holds the location of the proxy- and stub-code that must be used for the marshalling and unmarshalling process.

Figure 4 gives an example of a registry entry of a DCOM server-object. It gives the link between an APPID and the name of the executable to start.

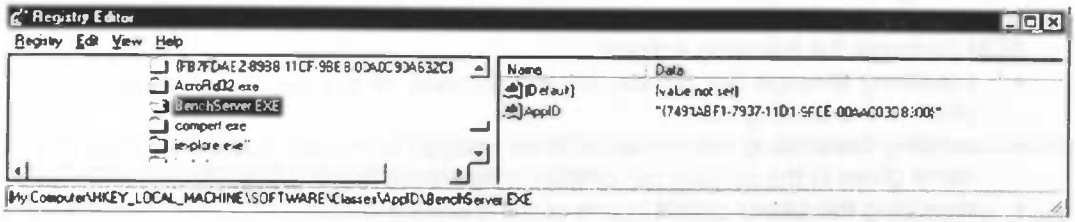


Figure 4:APPID registry entry

The following pictures give some more entries of the above mentioned executable. Figure 5 shows the entry for the proxy/stub-dll, which provides the marshalling and unmarshalling. and entries for the two classes that it exposes as server-objects.

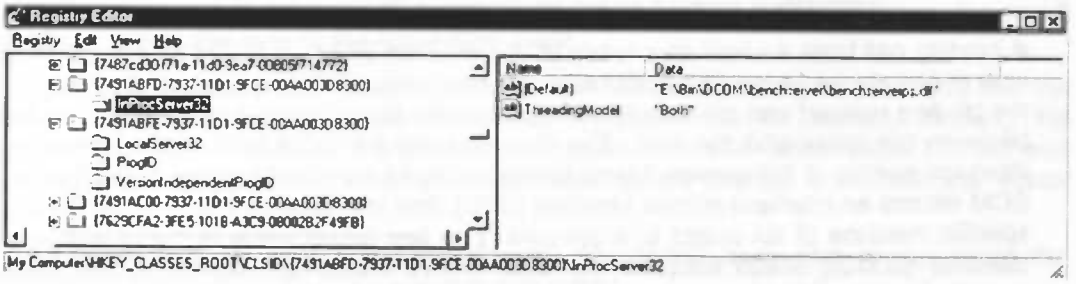


Figure 5:Proxy/stub registry entry

Figure 6 shows the entry for the CLSID and the connection to the right APPID.

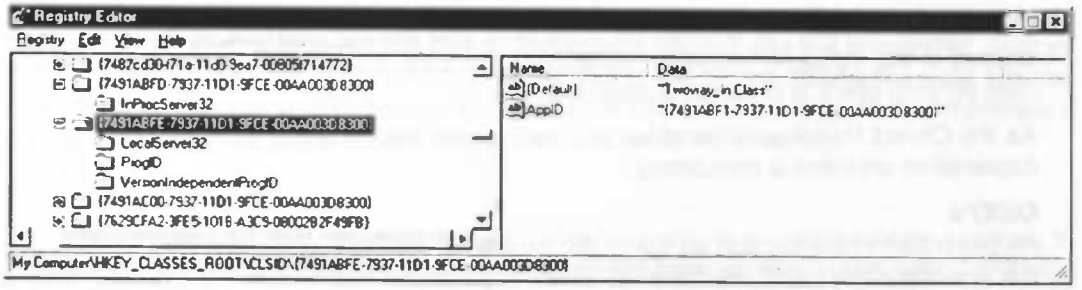


Figure 6:CLSID registry entry

Service Control Manager

The Service Control Manager (SCM also known as Scum) is the component that is responsible for locating, launching server-objects. The SCM is located on every DCOM enabled machine where clients and/or servers reside. The SCM is a RPC -server itself, and provides an important extension of DCOM over DCE RPC, in the way that it enables remote activation of server components. In DCE RPC a server component must be started prior to any binding or invocation attempts of a client. To enable this remote activation of DCOM-servers, the SCM exposes one interface, the IRemoteActivation-interface which has one method, the RemoteActivation method.

When a client does an attempt to bind to a server, there are several schemes possible to find the server, optionally start it and create the requested connection. As already seen in the previous section, several server types exist, but the client has also multiple possibilities when doing a request for a bind. In a client's call for a bind to an server, it can specify the type of server its wants:

- explicitly local
- explicitly remote
- no preference, either local or remote will do

In the latter two cases, the name of a remote computer-system can be applied on which the server should reside.

Depending on the information in the registry on the computer-system of the client, the parameters in the client's binding call and the type of server and its configuration, the SCM performs the following actions:

- searching through the registry for the location of the server-object with the CLSID given at the binding call
- sending the binding call to the SCM on another computer-system, with the computer-name given in the binding call or the configuration in the registry
- launching the server-object in one of the following ways:
    - loading and starting the server-object in the client's process space (in-process server)
    - loading and starting the server-object in its own process space (out-of-process server)
    - creating a surrogate and loading the requested server-object in the surrogate's process space (out-process DLL server)

A binding call from a client also leads to the creation of proxy-object on the client and a stub on the server (more on proxies and stubs will follow in the upcoming sections). When the binding request can be completed successfully, the SCM sets up an RPC connection between the proxy and the stub. The final task for the SCM is to return a marshalled interface pointer of the server-objects class-factory to the client's proxy. Furthermore, the SCM returns an interface pointer identifier (IPID), that identifies an interface belonging to a specific instance of an object in a process. The last return value is the object exporter identifier (OXID), which contains the binding information to connect to the interface specified by the IPID. The client now has a valid connection (i.e. an interface pointer) and can create the object it wanted and invoke requests on it.

Object Resolver

The Object Resolver is another RPC service, that closely interacts with the SCM and is located in the same NT-service, named RPCSS.EXE. Like the SCM, the Object Resolver runs on both client and server computers.

As the Object Resolver is involved in creating and manipulating so called OXID's, a brief explanation on them is mandatory.

OXID's

As described in the section on the SCM, an IPID is returned to a client. This IPID identifies the specific object and interface, to which a particular call should be directed. The IPID alone does not provide the necessary binding information to actually carry out a method-call. Information about server-addresses and underlying network protocols are also necessary. This binding information is stored in strings, the so-called string bindings. An Object eXporter IDentifier's (otherwise known as OXID's), is a variable that represents this binding information.

The Object Resolver performs the following services:

- providing the keep-alive mechanism that DCOM uses to keep its servers alive
- providing the resolving from OXID's to string-bindings

Client-side

Functions of the Object Resolver on the client-side:

- resolving of OXID's to string-bindings needed by the RPC-mechanism
- storing string bindings for local clients that connect to remote objects
- forwarding of unknown OXID's to appropriate server-machine
- pinging of server-objects
- delta-pinging for large number of references

Server-side

Functions of the Object Resolver on the server-side:

- resolving of OXID's to string-bindings needed by the RPC-mechanism
- storing string bindings for local servers
- resolving of OXID's unknown to clients
- forwarding of unknown OXID's to appropriate server-machine (middleman-function)
- rundown of servers or stubs in case of missing pings

- resolving of delta-ping protocol

Proxy and Stub

Shortly said, the proxy and the stub are used to marshal and unmarshal requests and parameters on behalf of the client and server-object.

A proxy is created by the COM-libraries when a client does a bind to a server. For the client the proxy is indistinguishable from the real server-object, because they behave equally. The function of the proxy is to provide the interface to the client and to remote the client's calls on the interface to the stub of the real server-object. When the client invokes a method on the server, this invocation is actually done on the proxy-object. The proxy-object marshals the call and the parameters and sends them to the stub on the other side of the communication channel. When the reply returns, the proxy unmarshals the result data and hands this over to the requesting client.

The stub for a server-object is build, when a client binds to that object. The stub has the actual references to the server-object, and is responsible for actually invoking the requests on the server-object an the clients behalf. The stub unmarshals the call and parameters coming from the client (marshalled by its proxy), and does the actual invocation.

They proxy and the stub communicate with the ORPC layer that encapsulates the underlying cross-process or cross-platform transport.

- The Proxy takes all the arguments from the client, converts them into the Network Data Representation (NDR) format, and generates the appropriate calls to the ORPC-layer.
- The Stub is located on side of the server, and maintains the real interface pointers to the object. When it receives the call, it unmarshals the call and the arguments, pushes the arguments on the stack and makes the call to the object. When the calls returns, the results are marshalled by the stub and sent back to the proxy, that unmarshals it returns and it to its client. For the client this process is invisible.

Client

The client is the part that requests the DCOM-infrastructure and some of its services to locate a server-object. Depending on the server-type a client requests, and the configuration stored in the registry, a server-object is started by the local or a remote SCM. A proxy object is also created, that serves as the local representation of the server-object. After a bind has been successfully completed, the client can invoke requests on the server-object without knowing the location of it and the effort that must be done to enable that.

By now the language support for writing clients is increasing, and even script-based languages can used to invoke requests on certain server-objects that enable this feature.

I(REM)Unknown

The IUnknown is an interface that every server-object must support. This interface is the entry-point for the pointer-tables that reference to the methods on the server-object. By binding to a server-object, a client gets a reference to this interface, which enables him to request a reference to the interface a client actually wants. With the introduction of DCOM, the IREMUnknown interface was introduced. This interface is requested in case of a remote reference by a client, and actually forwards all invocations on this interface to the IUnknown provided by the server-object. This mechanism enables the programming of just one interface, that doesn't need to know if the reference on the interface is local or remote.

2.3.4 Services

Microsoft has not a fixed set of defined services like CORBA has. This doesn't mean that there are no services, but they are provided as additional products that expose their functionality through a (D)COM interface. This creates a complete range of products that are plugged into the DCOM architecture and use it as its communication infrastructure.

The Microsoft Transaction Server (MTS) is completely built on DCOM and provides an addition to the DCOM architecture. It functions as a framework where simple server-objects can be used to create scale-able applications. Furthermore it provides a transaction monitoring and processing engine to safely use database transactions over multiple distributed databases.

The Message Queue Server (MQS) provides message oriented communication between clients and servers, which enables asynchronous communication.

Every product that Microsoft has developed the last few years, uses (D)COM as its communication infrastructure. This way, new services are added to DCOM as new server-side solutions and products are released.

2.3.5 Future enhancements

The future enhancements of DCOM are not very clear or outlined in a roadmap. The enhancements that will be made to DCOM will come one-by-one, and not as a release of a whole new specification.

With Windows NT5.0, active directories will become available. These active directories will provide a central repository for a group or domain of computers, in which much of the information that is stored in the registry now, will be stored. This also includes the configuration and registration of DCOM servers. This means that a client only has to ask the active directory for the location of a DCOM server.

Microsoft Clustering Server, is a way for server-farms, to provide services like load-balancing, fault-tolerance, replication or migration of server-objects. Although the use of these services will require Clustering Server-aware DCOM-servers, the mechanisms used to provide these functions will be based on DCOM.

2.4 A generic middleware architecture

To create a methodology for the performance evaluation of middleware, a generic architecture for middleware is created. This generic architecture is based on the decomposition of the two middleware products in the previous sections. The components of this generic architecture are drawn in Figure 7 and the functionality they must provide is described in the subsequent sections.

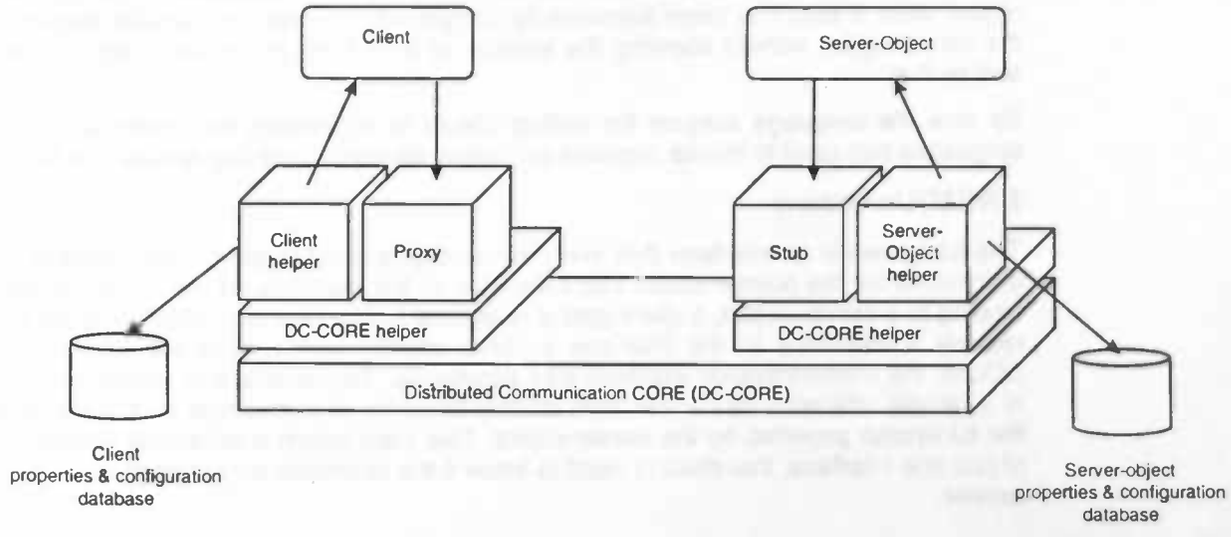


Figure 7:Generic middleware architecture

2.4.1 Generic middleware component description

This section gives a brief description of the functionality of the components that are identified to make up the generic middleware architecture.

Distributed Communication CORE (DC-CORE)

The Distributed Communication CORE is the component in the architecture that handles the actual transport of the invocation-requests from the client on the server-object. Furthermore, it shields the client and the server from the conversions of their message-exchange based communication to the actual packets that are transmitted by the used network-protocol.

DC-CORE helper

The DC-CORE helper is a component in the architecture that functions as an interface-layer between de communication core and the components that use this communication core. On the client-side is helps in addressing the correct server-object, while on the server-side it helps in decoding the incoming requests and delivering them at the correct server-object.

Client helper

The client helper is a component that helps the client while starting up and finding the services it needs. The information for configuring the client can be retrieved from the supporting database. This database holds all environmental information, so that the client can be started without needing any additional input. Hence, it helps in locating the requested server-object and issuing requests for binding to the that server-object.

Server-Object helper

The server-object helper is a component that helps the DC-CORE to communicate with the requested object. In the first place, it helps in locating and launching the server-object with information stored in the supporting database.

Properties & Configuration database

The properties and configuration database can provide storage for diverse parameters related with the client and server-object. Potential parameters can be:

- name of computer-system with a specific server-object
- launching and invocation authorisation
- environmental settings
- location of server-object on storage-device

Stub & Proxy

The stub and proxy provide the marshalling and unmarshalling of requests and their parameters. The proxy pretends to be the real object in perspective to the client, while the stub forwards the method-invocations, done by the client's proxy, to the real server-object.

Client

This is the actual requester of a service provided by one of the server-objects. It uses the proxy to invoke requests on a remote server-object through the DC-CORE. The client-helper is used to configure its environment and to locate the wanted server-object.

Server-Object

The server-object is the actual provider of services on the DC-CORE. It is launched on a client's request, or manually. The sever-helper, provides the necessary environmental settings for the server-object, while the stub invokes the actual requests on the behalf of the clients.

2.5 Conclusions

As a prerequisite for a performance evaluation of middleware products, knowledge of their internal architecture is needed to fully understand a performance evaluation and to interpret the results that are acquired during the measurements. To acquire this knowledge, the middleware products that are evaluated in this thesis (DCOM and CORBA), are examined and evaluated on their internal architecture. This knowledge about the internal architectures is used to create a generic middleware architecture. This generic architecture provides the foundation for the methodology of performance evaluating middleware presented in this thesis.

3 Benchmarking middleware

Performance evaluation and benchmarking of computer-systems, specific computer-parts and software running on these computer-systems are widely discussed topics. Many discussions deal with the method from which the performance numbers are obtained and what these numbers say about the system(part) that has been evaluated. In many cases 'magic' performance numbers arise without anybody knowing about the environmental details of the evaluation.

The importance of benchmarking middleware, is illustrated by the fact that at the moment of writing this thesis, the OMG has requested for proposals for ORB benchmarking. In the area of benchmarking distributed systems on high performance networks like ATM, the research of Gokhale and Schmidt [GOKHALE-97A][GOKHALE-97B] should be mentioned.

Before any performance evaluation can be executed, the primary targets should be clear and a suitable method must be chosen. Choosing a suitable evaluation method is not trivial, which will be made clear in the following sections. There are many techniques and combinations of methods for performance evaluation, suitable for different environments and situations.

This chapter also provides a nomenclature used throughout the rest of this thesis for describing client- and server-objects, the way these objects are organised and the computer-systems they run on. It provides a taxonomy for the different communication forms that are used when computational objects interact. Furthermore, it gives the measurement viewpoint used throughout the experiment descriptions and an identification of the possible bottlenecks.

One of the objectives of this chapter, is to give an overview on the basic performance evaluation techniques and their application fields. Another objective, and a very important one, is to develop a suitable technique that can be used in the performance evaluation of middleware. The chapter concludes with a section about the specific points a middleware performance evaluation should address and factors that may influence measurement data.

3.1 Introduction to performance evaluation techniques

In the quest to find a suitable performance evaluation technique, several basic techniques can be found. These techniques all have their own properties and are therefore used in specific application fields.

Performance evaluation is used in many application fields where a good knowledge is needed of a computer-system or specific part of it. In this context, not only hardware is considered, but also operating systems, specific applications or, like in this thesis, middleware. Typical application fields for performance evaluation are hardware and software design, selection, upgrade, tuning and analysis.

Performance evaluation techniques are used in many application fields and disciplines, therefore several techniques are being developed [KANT-92][WAL-90]. Despite the large number of available techniques, the following basic techniques can be identified:

- Measurement: Measurement is the most fundamental technique and its performance numbers are frequently used for other performance evaluation techniques. The performance numbers can e.g. expressed in the amount of latency induced by a certain component or like the amount of processing power needed for a computation. The acquired results can be used for the calibration of simulation and analytical models. Because measurements produce large amounts of raw data, the results must be statistically analysed to compute the outcome of the measurements.
- Simulation: This technique involves the creation of a model of the system being evaluated. Such a model is based on an architectural evaluation of the system, that can become parameters based on the outcome of previous measurements. A certain amount of input is applied to the simulated system to create a workload. Running a simulation produces, like measurements, large amounts of raw data that must be statistically analysed to acquire the desired results.
- Analysis: When an analytic modelling technique is used, a mathematical model of the system is constructed. Analytic models are only applicable when the system that has to be analysed is modelled in great detail. Because of the increasing complexity when the number of details increases, small systems or simplified systems are preferred.
- Hybrid techniques: In many cases a combination of the mentioned techniques is preferable over just using one technique. Different parts of the system can be evaluated with the most appropriate technique.

Irrespective of the technique used for the performance evaluation, certain inputs must be applied to the system or its model. These inputs must be handled by the system, and are commonly referred to as workload. Creating a workload specification is not trivial, as multiple difficulties arise about the nature of the workload. Possible points to consider are:

- when to perform a measurement
- how long to perform a measurement
- number of inputs / time-unit
- distribution of the inputs / time-unit or the measurement time
- composition of the different inputs

The composite of the points mentioned above, is called a *workload characterisation*. A workload characterisation builds only a model of the real workload, since not every aspect of the real workload may be equally important. The level of abstraction depends heavily on the used performance evaluation model. An analytical model usually requires a rather abstract workload (in terms of statistical properties) whereas a simulation can process a very detailed workload (like a trace of actual events).

Two different kinds of workload models can be identified, these are the executable and the non-executable model. An executable model, can be an actual trace, that is used as a direct input to do a simulation. Another example, is a program that generates inputs based on configurable parameters. A non-executable model, is a characterisation of the workload model, that can be used for analytic modelling.

With respect to performance evaluation, the following conclusions apply:

- middleware is through its distributed character, a very complex system. It consists of many components that can run on many computer-systems. Through the complexity, the analysis technique with its demand for modelling in great detail is not feasible.
- commercial middleware products must be used 'as is' (black-box principle), which means that reverse engineering is illegal. This makes a very detailed evaluation of its internal construction impossible. As a very detailed knowledge of the internal architecture is needed for a simulation-model, this technique is not suitable.
- as measurements can provide necessary modelling parameters for other performance evaluation techniques, and implementation details of middleware products are expected to influence the behaviour under heavy workloads, the measurement techniques is considered to be the most suitable technique to use for our purposes.

In the following sections we elaborate the measurement techniques for middleware.

3.2 Measurement techniques

When measuring a hardware or software system, interesting points are the state in which a system is, and the time that is spent in a certain state. To identify to current state of a system or a change in the state of a system, the following techniques can be used:

- sampled monitoring: On specified time-intervals, the state of the system(part) is checked and recorded.
- trace monitoring: On entry and exit points of important pieces of the system, events that change the state are recorded.

For measuring the time that is spent in a certain state, the above mentioned monitoring techniques can be applied using dedicated hardware, software, or a combination of both approaches:

- hardware monitoring: This technique employs additional monitoring hardware that is interfaced with the system under measurement in a non-intrusive way. The main advantage of this technique is that the measurement does not have to interfere with the normal functioning of the monitored system. The technique has the disadvantage that it is very expensive and that measurements on software is very difficult.
- software monitoring: This technique uses some measurement code either embedded in the existing software or as a separate set of routines. The main advantage is the generality and flexibility. The disadvantages are that it may seriously interfere with the normal functioning of the system and cannot be used for capturing fast occurring events. This technique is most appropriate for obtaining application and operating system related information, such as the time spent in a certain routine.
- hybrid monitoring: This technique uses the advantages of both hardware and software monitoring. All relevant signals are collected under software control and sent to another computer-system for measurement and processing. The advantages are that it is flexible and that its domain of application overlaps those both of hardware and software monitoring. Disadvantages are that synchronisation problems can occur between the measured system and the measuring system.

Hardware and hybrid monitoring are not suitable for measuring a complex distributed system, because of the complexity of the systems and the low-level techniques that must be used. Therefore, the software technique seems the most feasible technique for measuring the performance of middleware.

Instead of instrumenting (adding additional code-lines for software measurement) an existing application, a special piece of software, called a benchmark, will be used. Benchmarks are special pieces of software that are designed for performance measurement of a particular feature of the system.

3.3 Benchmarking, a measurement technique

A benchmark is a piece of software, where the software itself produces a certain workload and measures the performance of the particular part or system to evaluate. To get comparative results, the application domain for a benchmark should be characterised by a set of "typical" applications. These applications must be available on all the hard- and software systems on which the benchmark must be applied. Two ways can be identified for creating such a set of applications:

- choose a number of small applications which are used frequently and let them generate a certain workload. This is the so called 'application benchmark'
- create an artificial program with parameters that can be configured in such a way that it mimics a real application. This is the so called 'synthesised benchmark'

Both of these benchmarks types have their advantages and disadvantages. These are given in the following points:

- application benchmarks
    - advantages
        - use of real applications
        - use of script-based user-input
        - many applications available

    - disadvantages
        - only total application run times can be measured
        - no highlighting/pinpointing of specific parts
- synthesised benchmarks
    - advantages
        - exact highlighting/pinpointing of specific system-parts
        - very detailed measurements possible
        - programs can be created for all systems
        - free configurable parameters for workloads possible
    - disadvantages
        - misinterpretation of results through wrong measurement
        - possible influence of measurement while running
        - possibility of not correct 'working' programs through optimisations

As the purpose in this thesis is not only to evaluate the performance of middleware, but also to investigate its behaviour under a sustained stress of inputs to evaluate its scale-ability, the following requirements are made:

- the benchmark must be freely configurable to pinpoint performance bottlenecks
- the benchmark must be configurable to create a heavy and/or sustainable workload
- the benchmark has to be run on all middleware products evaluated
- the benchmark should create a workload that corresponds to real-world workloads

Considering the requirements mentioned above, and the advantages and disadvantages mentioned about application and synthesised benchmarks, the most suitable technique for our purpose, is the synthesised benchmark. In the next sections we will analyse how we can deal with the above mentioned identified disadvantages.

3.4 Benchmarking middleware

In the previous sections, performance evaluation techniques are discussed. The discussion concludes, that a synthesised benchmark is the most suitable technique to measure the performance of middleware platforms.

This means that a synthesised benchmark-suite must be developed, where the individual benchmarks are capable to address specific components of a middleware platform. The following points should be considered when developing the benchmark suite:

- measurement overhead should interfere as little as possible with the operation that must be measured
- create small benchmarks that address specific components or functionality's in the components of the middleware
- let the individual benchmarks be as lightweight as possible, to avoid influences from unnecessary system-load.
- create benchmarks that are easily portable, so they can be used on a wide range of middleware platforms
- let the individual benchmarks be freely configurable, so that the influences of different benchmark settings can evaluated
- provide realistic workloads during the benchmark-execution

3.4.1 Middleware request traversal

As already mentioned, the benchmark must be executed with the clients and server-objects on separate computer-systems. Furthermore, the response-time that must be measured, is the total response-time from a request-reply invocation from a client on a server-object. The following picture (Figure 8) gives an overview of the request-path that such a request must traverse and the components that it passes. This request-path is based on the generic middleware architecture developed in chapter 2, with the addition of the operating-system, hardware and network components.

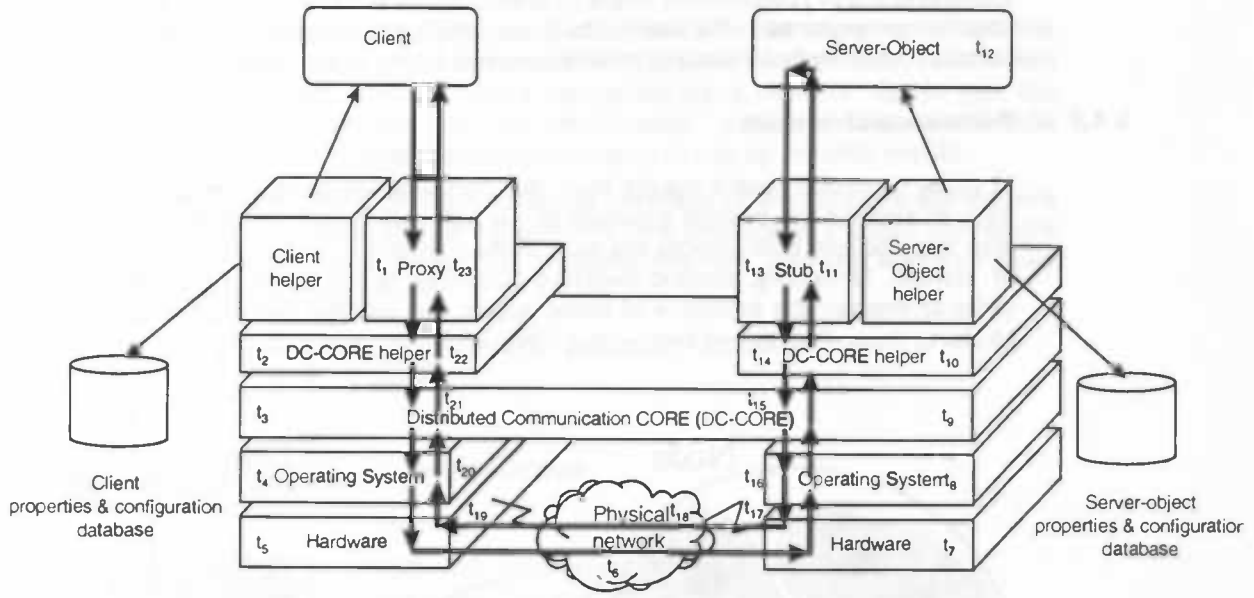


Figure 8:Request-path in middleware

The total response time in such a request can be considered as:

$$T_{res} = \sum_{n=1}^{n=23} t_n$$

In which t_n can be described as:

- t_1 = marshalling of request & parameters
- t_2 = preparation for distributed method call
- t_3 = Distributed method-call wrapper
- t_4 = Transfer of request through network stack
- t_5 = HW-level network transport of request
- t_6 = Physical network transport of request
- t_7 = HW-level network transport of request
- t_8 = Transfer of request through network stack
- t_9 = Distributed method-call un-wrapper
- t_{10} = Dispatching to correct server-object
- t_{11} = Unmarshalling of request & parameters
- t_{12} = Processing of request, producing result
- t_{13} = Marshalling result
- t_{14} = Directing to correct proxy
- t_{15} = Distributed method-call wrapper
- t_{16} = Transfer of result through network stack
- t_{17} = HW-level network transport of result
- t_{18} = Physical network transport of result
- t_{19} = HW-level network transport of result
- t_{20} = Transfer of result through network stack
- t_{21} = Distributed method-call un-wrapper
- t_{22} = Dispatching to correct proxy
- t_{23} = Unmarshalling result

Although many contributions to the total response time can be found, for the most of them it is not possible to measure their individually. Roughly speaking, the response time will be the sum of the following components:

- (de)marshalling of the parameters ($t_1 + t_{13} + t_{13} + t_{23}$)
- latency in network stacks ($t_4 + t_8 + t_{16} + t_{20}$)
- network latency ($t_5 + t_6 + t_7 + t_{17} + t_{18} + t_{19}$)
- dispatching of the request to the correct server-object (t_{10})

Identifying these components helps to create benchmark-descriptions that address these specific components in the architecture, or creating benchmark-descriptions that rule out certain components by making them a constant factor in the measurement.

3.4.2 Software architecture

In order to obtain useful results from the performance evaluation of the platforms, we have to analyse the internal structure of the software components. Moreover, we have to take into account how objects are executed on different nodes and using various threads of control. Obviously, context switches contribute to the performance of the system. In order to analyse the influence of these issues, we use the terminology of the Reference Model of Open Distributed Processing[1] (RM-ODP) [RAYMOND-97].

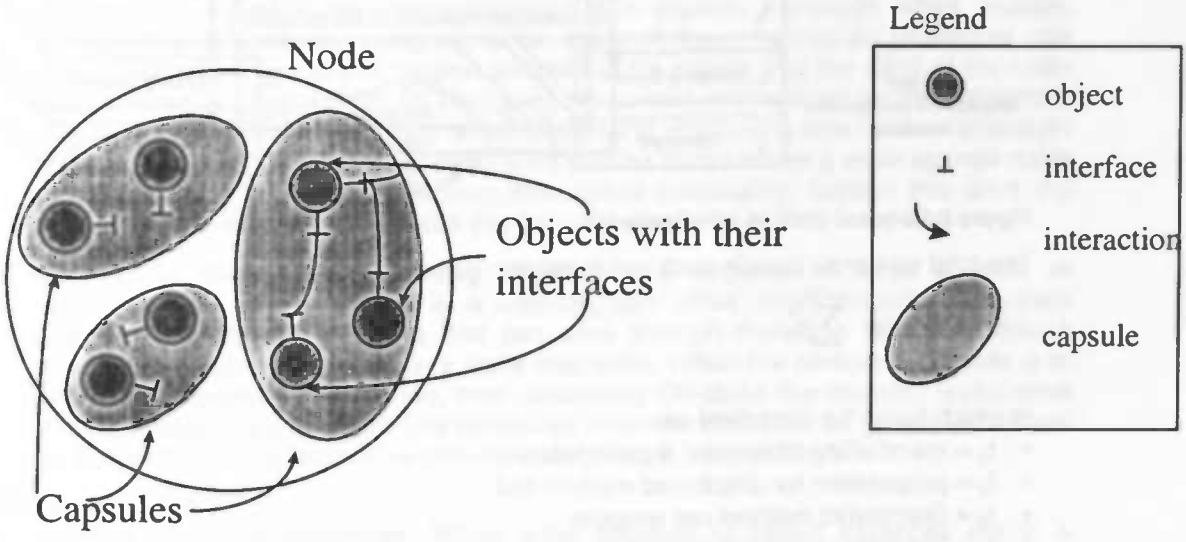


Figure 9:Taxonomy nodes, capsules and objects

In Figure 9, a distinction is made in nodes, capsules and objects. In this thesis the following description is used for them:

- a **node** is a computer system. It can be seen as the physical hardware and is also referred to as a host.
- a **capsule** is a configuration of engineering (or computational) objects forming a single unit for the purpose of encapsulation of processing and storage.
- an **object** is a model of an entity. An object is characterised by its behaviour and, dually, by its state. An object is distinct from any other object. An object is encapsulated, i.e. any change in its state can only occur as a result of an internal action or as a result of an interaction with its environment.

Due to this division in nodes, capsules and objects, the following different forms of communication between objects can be identified:

- **intra-capsule** interaction, is communication between objects belonging to the same capsule. This communication can be within one process or between different processes.
- **inter-capsule** interaction, is communication between objects belonging to different capsules. Capsules commonly reside in different processes, so inter-process communication will take place.
- **inter-node** interaction, is communication between objects residing on different nodes, and therefore automatically also in different capsules (capsules are bound to reside on one node).

[1] Although RM-ODP defines the concept of "clusters", this concept has been left out of the nomenclature used in this thesis. The defined distinction in nodes, capsules and objects is necessary for the upcoming experiment descriptions, the concept of "clusters" is not applicable here.

Although the communication between objects and capsules on the same node uses the infrastructure and technology that the middleware platforms provide, distributed computing (commonly) uses multiple nodes connected by a network. So to test the overhead and the performance of the middleware platforms, the inter-node communication is therefore the best communication type to use for realistic results.

Although the taxonomy of RM-ODP offers a broad coverage of the software architecture which is used in the description of the experiments, the concept of multiple lightweight processes in a capsule is not covered. To overcome this, the following additions have been made to the previous taxonomy:

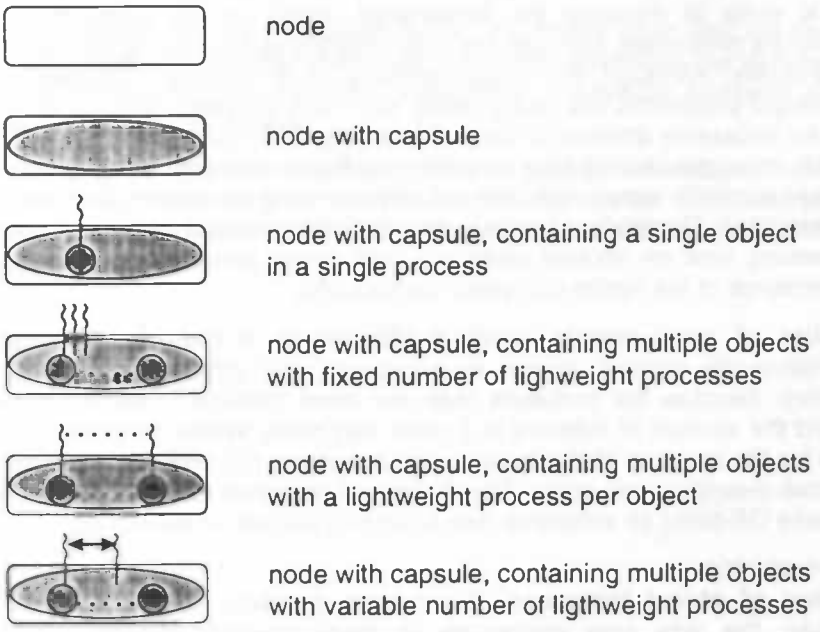


Figure 10:Extended taxonomy, with multiple lightweight processes

3.4.3 Benchmarking viewpoint

The experiment descriptions and benchmark specifications in the following sections, are based on a client's viewpoint. This means that the performance is evaluated from the viewpoint of the latency a client experiences. To enforce this viewpoint, a typical client-server set-up is chosen for the experiment descriptions. This means that there is one server-node on the network on which a capsule resides that exposes a certain service. The other nodes have clients that invoke requests on this service from the server-node. The performance is evaluated by measuring the response-time (or maybe waiting-time) a client experiences before the request returns.

Because this thesis describes a methodology for measuring the performance of the middleware, the data that is used in the requests is meaningless and there won't be any processing on the server-side of the benchmark. This means that the server returns a request as soon as possible.

With these two assumptions, the experiments evaluate the latency a client experiences when using a service that is located on a remote node.

3.4.4 Controlling the external factors

Before describing the actual experiments in the following chapter, it is useful to identify external factors that may become possible bottlenecks. First the possible bottlenecks that can occur in general are described and furthermore the bottlenecks specifically for server-nodes.

Possible general bottlenecks:

- **maximum network-load**. The network that connects the computer-systems on which the benchmark is executed, can have different topologies and transport-speeds. Depending on the speed of the network, the topology and the placement of the computer-systems executing the benchmark, these factors can have a great influence on the measurements. As many networks are based on technologies in which multiple nodes can do network accesses simultaneously[2], the problem can arise that the network becomes overloaded. This phenomenon should be monitored as the resulting measurement-data can be greatly affected.
- **overhead due to context-switching**. The capsules in the client-nodes and the objects in the server-node may use multiple lightweight processes. On the client-node this is done to increase the server-load, while on the server-node the multiple lightweight processes increase the responsiveness for serving the requests from the client-nodes. Although the responsiveness of a capsule increases when multiple lightweight processes are being used, too many of these lightweight processes can give an excessive amount of context switches. This means that the CPU of the node spends more processing time on setting up the correct environment for the lightweight process currently scheduled, than actually spending processing time on the lightweight process itself. Generally a break-even point can be found where a node spends more processing time on context switching then actual processing, beyond this point the performance of the nodes collapses dramatically.
- **number of concurrently running objects in a capsule**. A large number of simultaneously running objects in a capsule, can utilise large amounts of system memory. Besides the problems that can arise through threading issues, problems around the amount of memory in a node may arise. When the number of objects is to large for the memory of the node, time consuming OS-tasks like memory reallocation and disk-swapping can occur. The measured response times will be greatly influenced by these OS-tasks so extensive disk-swapping should be monitored.

For a server-node:

- **number of object instances**. When large numbers of object instances run in a capsule, the way how references to these running objects are stored in the middleware-layer can be a great influence on the performance. When an object is identified with a number, and the references the objects are kept in a list, for the object that is the last in that list, the access-time will be considerably longer as that of the first object in the list. Other storage strategies, like hashing or binary trees, for the references will give a constant access time.
- **the list of pending incoming requests**. When the number of incoming requests is larger than the number of simultaneously running objects, the request must be stored until it can be served. This storage of requests is generally done with a 'pending request list'. This list should be as short as possible, but due to too much incoming request from the client-nodes this list can grow rapidly. When this list grows until an unacceptable size (more than the number of simultaneously running objects), this indicates a lack of performance on the server-node or a shortage of simultaneously running lightweight processes.

In summary, the following attention points can be identified, that must be monitored during the execution of the experiments, in which they should not become the bottleneck, because that would influence the results of the measurements.

- the load on the network
- overhead due to excessive context-switching
- memory-load of the nodes

---

[2] Widely spread are networks based on CSMA/CD technology (Carrier Sense, Multiple Access with Collision Detection), like Ethernet.

3.5 Conclusions

In this chapter we have identified the various available methods and techniques for performance evaluation of middleware implementations. We conclude that middleware can be evaluated using measurements by using software implemented monitoring techniques based on synthesised benchmarks. Furthermore, we identified various external factors that have to be controlled. Care have to be taken when designing a model for modelling the right system parts with the correct parameters and measurement data should be treated with a little suspicion about the absolute truth and objectivity of the results.

The software systems that are evaluated during this thesis must and will be seen as a black box. This approach is taken, because this thesis describes the development of a benchmarking methodology for standard "of-the-shelf" middleware products, and reverse-engineering these products is illegal.

This chapter also provided a taxonomy for the various interaction forms between distributed objects, which is used in the rest of this thesis. a description of the components a request must traverse during a remote invocation and an identification of possible bottlenecks.

4 Experiments and benchmarks

The previous two chapters provide a generic middleware infrastructure and a way for measuring the performance of commercially available middleware products. The presented measurement technique is the synthesised benchmark, but as already stated, a synthesised benchmark must address specific problem areas in the system that is evaluated.

As one of the objectives is to describe a methodology for performance evaluation and scale-ability measurements of middleware products. To accomplish this, this chapter identifies a set of experiments, that can be conducted for such a performance evaluation at specific problem areas.

An implementation of several benchmarks is given for CORBA and DCOM. These benchmarks are actually executed on the KPN Research office-network, of which the set-up and the results are given. This office-network provides the means to actually run large scale experiments with up to hundreds of computing systems. During the experiments we have limited the number of participating computing systems to fifty.

The objectives of this chapter are therefore a description of a set of experiments, an experiment set-up and a description of possible benchmarks to create a comprehensive benchmark-suite. A subset of these experiments are executed on two middleware products, to evaluate their performance and the usability of the methodology.

4.1 Experiment descriptions

To develop experiment descriptions that highlight specific components of the middleware, so that the measurements unveil there behaviour, the following strategy is used:

- identify a functionality of a component that can be identified and preferably an request which specifically puts high demands on this component in the middleware architecture
- create a specific request-type that invokes this operation
- create an invocation-scenario in which the behaviour of the specific component is tested on its performance behaviour.

Besides this strategy, the following constraints on the environment and configurations where taken in consideration, during the development of the experiment descriptions:

- every node will host one capsule
- a capsule may host one or more objects
- a capsule may use one or more lightweight processes
- an experiment can comprehend one or more client-nodes
- there will always be one server-node

While formulating the experiment descriptions based on the given strategy and constraints, a repeating pattern was identified. This pattern consists of a constant increase of client-nodes, server-object instances and client-object instances. Based on this pattern, it is possible to create three matrices, which are presented in the following sections. Each of these matrices, describes four possible experiments.

4.1.1 Experiment matrix 1

The first matrix (Figure 11) describes several experiments where one server-node and one client-node are used. On the client-node the experiments will scale from 1-to-n objects per capsules. Each of these objects will run on its own lightweight process. On the server-node there will be one capsule that contains 1-to-m objects. For improvement of the responsiveness, the objects will run on multiple lightweight processes (not necessarily a lightweight process per object).

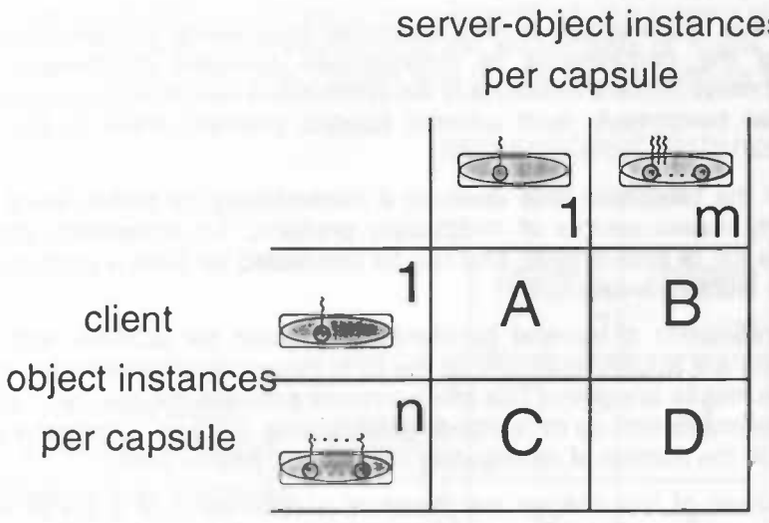


Figure 11:Experiment matrix 1

Short explanations of the experiments of matrix 1:

A) This experiment has a 1-on-1 relation. The goal of this experiment is primarily intended to measure the middleware communication performance. This benchmark will conduct measurements that focus on middleware overhead and the marshalling of the data, which is performed by the proxy and stub of the generic architecture, described in chapter 2.

B) This experiment has a 1-on-m relation. The goal of this experiment is primarily intended to measure the middleware overhead in dispatching a client's request for a specific object instance within a large (m) number of object instances in the server capsule, which is performed by the DC-CORE helper on the server-side. The dispatching overhead can be measured by examining the response times for accessing the different objects on the server-node.

C) This experiment has an n-on-1 relation. The goal of this experiment is to find the maximum number of objects in one capsule that can 'simultaneously' run and reside on one client node. This maximum can be found by examining the memory load on the client-node, the load of the server-node and the network-traffic.

D) This experiment has an n-on-m relation. The goal of this experiment is to examine the behaviour of the middleware when a client with n objects (for example, the value for n can result from experiment C) invokes requests on an increasing number of server-object instances, using multiple lightweight processes..

4.1.2 Experiment matrix 2

The second matrix (Figure 12) describes several experiments where one server-node and 1-to-k client -nodes are being used. On the client-node the experiments will use one object per capsule. The objects will each run on their own lightweight process. On the server-node there will be one capsule that contains 1-to-m objects. For improvement of the responsiveness, the objects will run on multiple lightweight processes (not necessarily a lightweight process per object). These experiments are meant to evaluate the behaviour of the DC-CORE and the DC-CORE helper at the server-side, when a large number of client-nodes invoke requests on a server-node in which the number of objects and lightweight processes is increased.

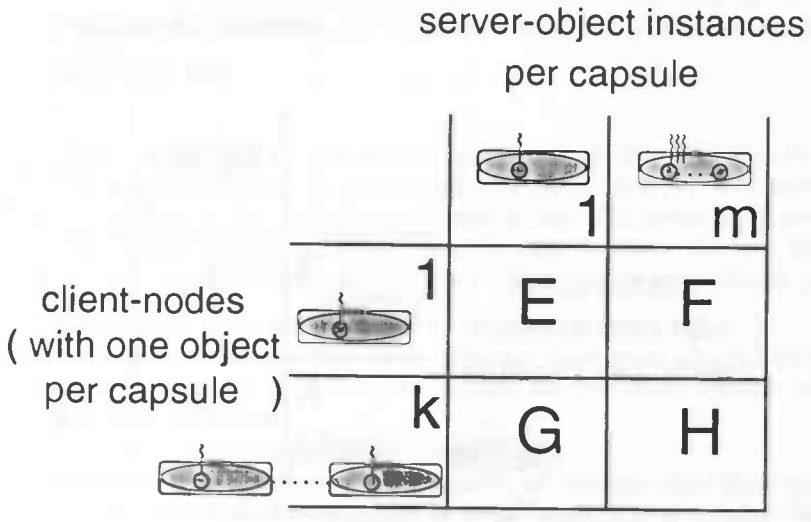


Figure 12:Experiment matrix 2

Short explanation of the experiments of matrix 2:

E) This experiment has a set-up that is comparable to that of experiment A, but must be executed on the same network infrastructure as the experiments with large numbers of client-nodes. The benchmark algorithm that will be used here, is optimised for scale-ability experiments and not for raw communication performance. This experiment won't provide extra valuable information on marshalling overhead, but the resulting data is used as a reference (like a zero-measurement) for the other experiments.

F) This experiment has a set-up that is comparable to that of experiment B but must be executed on the same network infrastructure as the experiments with large numbers of client-nodes. The benchmark algorithm that will be used here is also not specifically designed for measuring the dispatching of requests to large numbers of objects. Although the resulting data is used as a reference for other experiments.

G) This experiment has a k-on-1 relation. The goal of this experiment is to find the relation of the response time and the number of client nodes that are invoking requests on a single server-node with one object-instance in its capsule.

H) This experiment has a k-on-m relation. The goal of this experiment is to find the relation of the response time and the number of client-nodes that can invoke requests on a single server capsule, and thereby finding the optimum number of object-instances and lightweight processes in the server-capsule to handle a high load of client requests.

4.1.3 Experiment matrix 3

The third matrix (Figure 13) describes several experiments where one server-node and 1-to-k client-nodes are being used. On the client-node the experiments will use 1-to-n objects per capsule. The objects will each run on their own lightweight process. On the server- node there will be one capsule that contains 1-to-m objects[3]. For improvement of the responsiveness, the objects will run on multiple lightweight processes (not necessarily a lightweight process per object).
These experiments are also meant to evaluate the behaviour of the DC-CORE and the DC-CORE helper at the server-side, but in these experiments the number of client objects on every node is increased. This extension to the experiments of matrix 2 is made to increase the load on the DC-CORE helpers and the DC-CORE, without increasing the number of client-nodes.

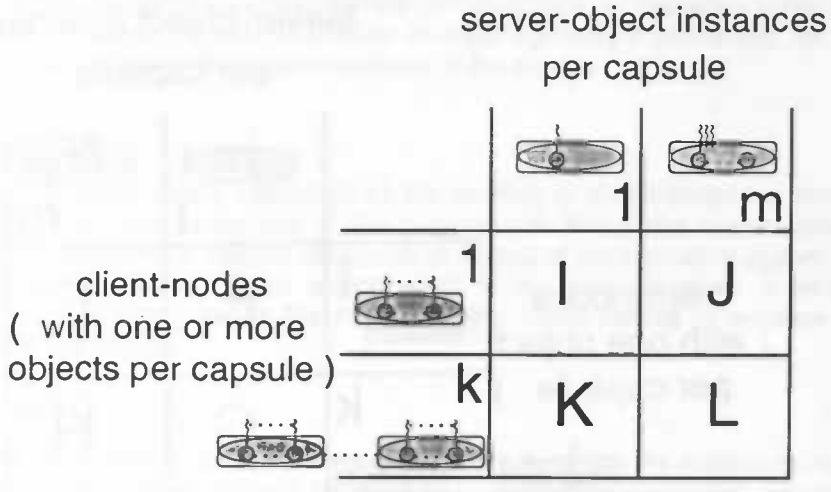


Figure 13:Experiment matrix 3

Explanation of the experiments of matrix 3:

K) The experiment has a n-on-k-on-1 relation. The goal is to examine the behaviour of the middleware when an increasing number of client capsules, each with n objects (n can be the result of exp. C) invokes requests on a server capsule with a single object instance.

L) The experiment has a n-on-k-on-m relation. The goal is to examine the behaviour of the middleware when an increasing number of client-capsules, each with n objects (n can be the result of exp. C) invoke requests on a server capsule. This server capsule has m object instances, where the goal is to find the optimum number of instances and lightweight processes to handle the requests.

[3]When a experiment describes multiple objects in a server capsule, the reader should be aware that the object runs on multiple lightweight processes within the capsule. This requires the capsule to actually handle multiple request in parallel, instead of only allocating memory for the objects and handle the requests sequentially.

4.2 Detailed experiment descriptions

In the previous paragraph, a total number of ten experiments where identified. To obtain comparable measurement results, the following sections describe the experiments in more detail. It also addresses the application and use of the experiments, and the specific components of the middleware architecture they address.

4.2.1 Experiments of matrix 1

The experiment descriptions given in this section, are the experiments belonging to the first matrix, and comprehend experiment A, B, C and D.

When the experiments described here are implemented, it will lead to a couple of benchmarks that are specialised in the performance evaluation of a particular function of the middleware platform.

Experiment A

Experiment A can be considered as the experiment, in which the middleware overhead and the marshalling performance is evaluated. Because this is a communication benchmark, the information of the data is meaningless, and the processing overhead on the server-side should be a low as possible. This will actually mean that the data is only consumed by the server, and that the result data is predetermined or randomly generated.

To conduct this experiment, a benchmark must be developed that:

- invokes request with parameters that have different data-types (from basic types to complex structures). The set of data-types, should be the cross-section of the IDL-types of the evaluated platforms.
- invokes requests with increasing parameter-sizes.
- uses parameter-sizes that result in equal amounts of network-data (this reduces the influence of the network-stack, the hardware and the network to a constant factor while the data-types can varied).

Although the two basic variables to test are mentioned in the description, some environmental changes can also be used. One could think of the following:

- in- or decreasing the processor speed, for influencing the TCP/IP stack and operating system latency.
- Using different NIC's (Network Interface Cards) and corresponding network-infrastructures for determining the influence of the network latency.

Attention-points

- maximum network-load: as the size of the parameter increases, the influence of the network latency and throughput is expected to become a large part of the total response-time.

  As an example, consider a parameter-list of 200Kbytes, and a 10Mbit network. Such a network can practically transport one Mbyte of data per second. For the transport of the 200Kbytes, it takes at least 200Ms for the network transport.

Experiment B

Experiment B can be considered as the experiment, in which the dispatching performance of the stub and the DC-CORE helper on the server-node is evaluated. With dispatching is meant, the addressing of the correct server-object within a capsule when a large number of server-objects are running.

Depending on the platforms that are evaluated, there are two approaches possible to access a particular server-object within a large collection:

1) named instances. This means that a server-object can be given a name, by which it can be identified. When this technique is supported, the server should create a large number of server-objects while giving them a well-known name. The objects can be named by their creation-number. The client can bind to a few of such named server-object, and hold a reference to it.

2) large arrays with server-object references. This means that server-objects can only be identified by the reference a client gets when it binds to the server-object. To create a large number of these objects and to invoke requests on a random server-object, a client needs to hold references to all the server-objects it may possibly address.

If possible, the more elegant strategy 1 should be used in favour to strategy 2. In every case, the strategy used for a comparison between multiple platforms should be the same.

To conduct this experiment, a benchmark must be developed that:
- creates a large number of server-objects in increasing quantities per test-run.
- invokes requests on evenly distributed server-objects, between the first and last one that are created.
- uses several invocation-schemes to test or detect possible caching algorithms when referencing the same server-object multiple times.
- uses a single lightweight process for the server-objects, to decrease context-switching overhead. A server-capsule that uses multiple lightweight processes for its server-objects, won't improve the responsiveness of the server.

Attention-points

- number of server-object instances: as the number of object-instances increases, the server needs more resources of the server-node. When the server-node is out of available resources, it causes excessive overhead of the operating system for freeing up resources. Accessing an object which is not referenced for a while can be temporarily swapped out of the main memory, which results in extreme response-times for this object.

Experiment C

Experiment C can be considered as the experiment, in which the maximum number of n concurrently running client-objects in one client-capsule is determined, as more than n client objects would severely decrease the response-time. This means that multiple lightweight processes must be created and that every lightweight process behaves as a complete client-process.

To conduct this experiment, a benchmark must be developed that:
- has a client that creates a configurable number of lightweight processes.
- behaves as an independent client-process in each lightweight process.
- lets every independent client-process invoke requests on a single server-object.

Problems that can arise during the execution of such a benchmark, like a maximum on the number of lightweight processes, can be introduced by the operating system.

Attention-points

- maximum network-load: as a large number of client objects invoke requests on the server, the number requests can become larger than the number of packets the network can transport. (every request-reply pair will use at least two network packets)
- context-switching overhead: due to the possibly large number of concurrently running client-objects, the overhead in switching from one lightweight process to the other can cause excessive overhead, in which the node has no time to serve an client-object but is just switching from one to the other.
- number of simultaneously running client-objects: as the number of client-objects grows, the resources on the node may run out, in which case the operating system starts freeing up resources which is undesirable during the measurements

Experiment D

Experiment D can be considered as the experiment, in which the behaviour of the middleware is examined when a varying number of server-objects and/or lightweight processes used to serve the requests of a single client-capsule. This client-capsule is run with as many multiple lightweight processes as possible.

To conduct this experiment, a benchmark must be developed that:

- creates a configurable number of server-objects.
- creates a configurable number of lightweight processes, on which the server-objects server the incoming requests.
- creates a configurable number of lightweight processes, so that the client-capsule can concurrently invoke requests on the server-capsule.

Attention-points

- maximum network-load: as a large number of client objects invoke requests on the server, the number requests can become larger than the number of packets the network can transport. (every request-reply pair will use at least two network packets)
- context-switching overhead: due to the possibly large number of concurrently running objects, the overhead in switching from one lightweight process to the other can cause excessive overhead, in which the node has no time to serve an object but is just switching from one to the other.
- number of simultaneously running client-objects: as the number of client-objects grows, the resources on the node may run out, in which case the operating system starts freeing up resources which is undesirable during the measurements
- number of server-object instances: as the number of object-instances increases, the server needs more resources of the server-node. When the server-node is out of available resources, it causes excessive overhead of the operating system for freeing up resources. Accessing an object which is not referenced for a while can be temporarily swapped out of the main memory, which results in extreme response-times for this object.

4.2.2 Experiments of matrix 2

The experiments defined in this matrix, are meant to evaluate the behaviour of the middleware when it is exposed to a large number of client-nodes invoking requests on a single server-node. This behaviour is mainly dependent on the DC-CORE and the DC-CORE helper at the server-side. By using multiple client-nodes, the behaviour is measured when the mentioned components must handle requests coming from these multiple client-nodes. The two experiments E and F, give the lower-bound response times for the configurations of the server-capsules and the used network. The components of the middleware architecture that are measured here

Experiment E

Experiment E can be considered as the experiment, in which a ground-level experiment is conducted for the use of the scale-ability benchmarks. While this experiment uses only one client-capsule, and one server-object in the server-capsule, this experiment gives the lower-bound in response-times for scale-ability benchmarks.

To conduct this experiment, a benchmark must be developed that:

- has a client that creates a single lightweight process.
- has a single request with a very short parameter-list, like a single integer.
- has a server with one server-object on a single lightweight-process.

Attention-points

- network-load: as this experiment uses only one client-object and one server-object, monitoring the network-traffic on packets and throughput give the network-load as induced by one client-object.

Experiment F

Experiment F can be considered as an ground-level measurement for the rest of the experiments with multiple server-objects in the server-capsule. While this experiment uses only one client-capsule, and multiple server-objects in the server-capsule, this experiment gives the lower-bound in response-times with this kind of server-capsules.

To conduct this experiment, the following modification to experiment-description E must be made:

- the server-capsule should have more than one server-object, which are run on multiple lightweight processes.

Attention-points

- network-load: as this experiment uses only one client-object and a server-capsule with multiple server-objects and or lightweight processes, monitoring the network-traffic on packets and throughput give the network-load as induced by one client-object.
- context-switching overhead: due to the possibly large number of concurrently running objects in the server-capsule, the overhead in switching from one lightweight process to the other can cause excessive overhead, in which the node has no time to serve an object but is just switching from one to the other.
- number of server-object instances: as the number of object-instances increases, the server needs more resources of the server-node. When the server-node is out of available resources, it causes excessive overhead of the operating system for freeing up resources. Accessing an object which is not referenced for a while can be temporarily swapped out of the main memory, which results in extreme response-times for this object.

Experiment G

Experiment G can be considered as the measurement, in which the behaviour of the middleware is evaluated when multiple client-nodes invoke requests on the server-node. The behaviour is based on measuring the response-times the clients experience. The server set-up is equal to the set-up of experiment E.

The objective of this experiment is to create an increasing load on the middleware and the server-capsule, while knowing that the client-nodes have enough resources for serving the client-objects immediately. Because this experiment involves multiple client-nodes and to create such a load during a reasonable time, it is necessary to synchronise all the client-capsules, so that they start their invocation requests simultaneously.

To conduct this experiment, a benchmark must be developed that:
- has a synchronisation-service on which the clients can synchronise for executing their invocation requests on the server-node.
- has clients that connect to a synchronisation-service for their synchronisation and to the server-node on which the invocation-requests are being made.

Attention-points

- maximum network-load: as multiple client-nodes invoke requests on the server-node, the number of packets that the network can transport can be a limiting factor. When this maximum is reached, the response-times that are measured are not depending on the middleware anymore, but depend on the maximum number of network packets that can be transported on this particular network-infrastructure.

Experiment H

This experiment is intended to evaluate the behaviour of the middleware when the invocation-requests of the client-nodes, are being served by a server-capsule with multiple object instances and/or multiple lightweight processes. This experiment shows the influence of a server-capsule that serves requests simultaneously, where the amount of parallelism is determined by the number of lightweight processes, in respect with a server-capsule that handles every incoming requests sequentially.

To conduct this experiment, use:
- the clients and synchronisation service of experiment G
- the server-capsule of experiment F

Attention-points

- maximum network-load: as multiple client-nodes invoke requests on the server-node, the number of packets that the network can transport can be a limiting factor. When this maximum is reached, the response-times that are measured are not depending

on the middleware anymore, but depend on the maximum number of network packets that can be transported on this particular network-infrastructure.
- context-switching overhead: due to the possibly large number of concurrently running objects in the server-capsule, the overhead in switching from one lightweight process to the other can cause excessive overhead, in which the node has no time to serve an object but is just switching from one to the other.
- number of server-object instances: as the number of object-instances increases, the server needs more resources of the server-node. When the server-node is out of available resources, it causes excessive overhead of the operating system for freeing up resources. Accessing an object which is not referenced for a while can be temporarily swapped out of the main memory, which results in extreme response-times for this object.

4.2.3 Experiments of matrix 3

The experiments presented in matrix 3, extend the experiments of matrix 2 by client-capsules that have multiple lightweight processes, which all behave as client processes. These experiments have been defined to increase the load on the middleware and the server-capsule, which is more realistic than a single client per node. The components that will be set under a high load, are the DC-CORE and the DC-CORE helpers.

Experiment K

This experiment extends experiment G, by using multiple lightweight processes in the client-capsule to increase the stress applied on the middleware and the server-capsule. The client-capsules, therefore should have a configurable number of lightweight processes.

To conduct this experiment, the following benchmark must be developed:
- the client-capsule from experiment C, with the client-synchronise facilities and synchronisation service introduced in experiment G.
- the server-capsule from experiment J.

Attention-points

- maximum network-load: as multiple client-nodes invoke requests on the server-node, the number of packets that the network can transport can be a limiting factor. When this maximum is reached, the response-times that are measured are not depending on the middleware anymore, but depend on the maximum number of network packets that can be transported on this particular network-infrastructure.
- context-switching overhead: due to the possibly large number of concurrently running objects in the client-capsule, the overhead in switching from one lightweight process to the other can cause excessive overhead, in which the node has no time to serve an object but is just switching from one to the other.
- number of simultaneously running client-objects: as the number of client-objects grows, the resources on the node may run out, in which case the operating system starts freeing up resources which is undesirable during the measurements

Experiment L

This experiment extends the load on the mentioned components in experiment K by using multiple server-objects and/or multiple lightweight processes in the server-capsule to handle the incoming invocation-requests.

To conduct this experiment, the following benchmark must be developed:
- the client-capsules and synchronisation service used in experiment K.
- the server-capsule introduced in experiment H.

Attention-points

- maximum network-load: as multiple client-nodes invoke requests on the server-node, the number of packets that the network can transport can be a limiting factor. When this maximum is reached, the response-times that are measured are not depending on the middleware anymore, but depend on the maximum number of network packets that can be transported on this particular network-infrastructure.
- context-switching overhead: due to the possibly large number of concurrently running objects in the client-capsule, the overhead in switching from one lightweight process to the other can cause excessive overhead, in which the node has no time to serve an object but is just switching from one to the other.
- number of simultaneously running client-objects: as the number of client-objects grows, the resources on the node may run out, in which case the operating system starts freeing up resources which is undesirable during the measurements
- number of server-object instances: as the number of object-instances increases, the server needs more resources of the server-node. When the server-node is out of available resources, it causes excessive overhead of the operating system for freeing up resources. Accessing an object which is not referenced for a while can be temporarily swapped out of the main memory, which results in extreme response-times for this object.

4.3 Evaluating CORBA & DCOM

The previous sections give the experiments and benchmark descriptions to create a benchmark-suite, to evaluate components in middleware and the response-times a client experiences when the middleware is exposed to high loads.

This section gives the implementation details of some of the experiments that are conducted on Microsoft's DCOM and the CORBA implementation Orbix from IONA Technologies. A part of all the experiments described in section 4.1 and 4.2, where implemented due to the available time. Although not every benchmark is implemented, a reasonable part of the benchmark suite is developed and executed, to evaluate the two middleware implementations.

The objective of this section is, to give an example-implementation of the descriptions given in the previous sections, to describe the test and measurement environment and finally present the interpreted results of the benchmark-suite.

4.3.1 Hardware & software environment

This section gives a description of the hardware and software used, during the development and execution of most of the benchmarks. When an experiment is executed on a different environment, the environment is shortly described in the section about that particular benchmark.

All benchmarks were executed on standard Intel x86 based PC's running Microsoft Windows NT 4.0.

Development environment CORBA:
- Orbix version 2.3 (Iona Technologies)
- Microsoft Visual C++ 5.0 Professional edition (with service pack 3)

Development environment DCOM:
- NT 4.0 service pack 3
- Microsoft Visual C++ 5.0 Professional edition (with service pack 3)

Server-node:
- Dual Intel Pentium II 266Mhz processor (SMP-configuration)
- 128 Mb internal memory
- Windows NT Server 4.0 (with Service Pack 3)

- Microsoft Option Pack, with following parts installed
  - Internet Information Server 4.0
  - Transaction Server 2.0
- Orbix version 2.3
- 10/100Mbit NIC

Client-nodes
- Intel Pentium Pro 200Mhz processor
- 32 Mb internal memory
- Windows NT Workstation 4.0 (with Service Pack 1)
- 10Mbit integrated NIC

Network configurations:

For the benchmarks with single client-node, a separate hub (network concentrator) is used to rule out unwanted influences of the office-network.

The network used during benchmarks with multiple client-nodes, is the office network of KPN-Research in Groningen. The network is based on a multi-segment UTP-network. UTP is an acronym for unshielded twisted-pair, which gives a star-based network topology. The experiments are all executed on networks with low or no other network traffic. This approach is chosen, to decrease actual influences on the measurements.

Network configuration for Experiment A

The following figure gives the network configuration used during Experiment A. This configuration is also suitable for executing Experiments B,C and D

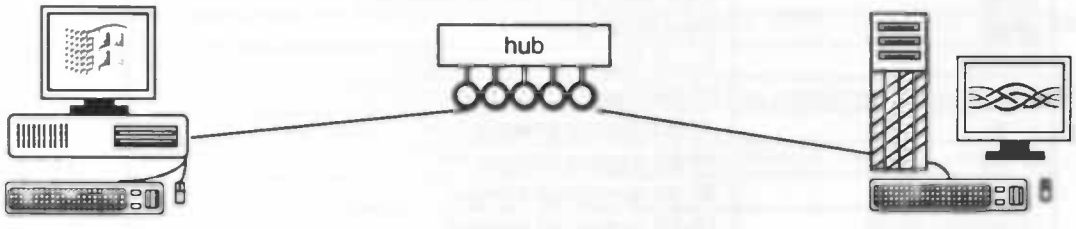


Figure 14:Two-node network-configuration

4.3.2 Experiment A

Implementation

The experiment description gives the following rules for developing an appropriate benchmark:
- invoke requests with parameters that have different data-types (from basic types to complex structures). The set of data-types, should be the cross-section of the IDL-types of the evaluated platforms.
- use parameter-sizes that result in equal amounts of network-data for every data-type used as parameter (this reduces the influence of the network-stack, the hardware and the network to a constant factor when different data-types are used).
- invoke requests with increasing parameter-sizes.

IDL-definition for marshalling benchmark

This section describes the types that can be used in the IDL's of both middleware implementations. As the functionality of the benchmark on both middleware platforms must be equal, the IDL types that will be used for the benchmark IDL must be supported in both IDL's.

Basic IDL-types for MIDL:

Base Type	Description
boolean	8-bit data item

byte	8-bit data item (no conversion)
char	8-bit unsigned data item
double	64-bit floating-point number
float	32-bit floating-point number
handle_t	Primitive handle type
hyper	64-bit signed integer
long	32-bit signed integer
short	16-bit signed integer
small	8-bit signed integer
void*	32-bit context handle
wchar_t	16-bit unsigned data item

For DCOM-IDL there are several compound-types:

- There are two types of `arrays` (both uni- or multi-dimensional):
  - bound, defined by a type and with a length known at compile time;
  - unbound, only defined by the data-type and a length-identifier that is known at run-time;
- And two types of `strings` available:
  - bound, with fixed length known at compile time
  - unbound, known length at run-time.

Basic IDL-types for OMG-IDL:

Base Type	Description
boolean	8-bit data item (TRUE or FALSE)
octet	8-bit data item (no conversion)
char	8-bit data item
double	64-bit floating-point number
float	32-bit floating-point number
long	32-bit signed integer
short	16-bit signed integer
unsigned long	32-bit unsigned integer
unsigned short	16-bit unsigned integer
any	allows the specification of values that can express an arbitrary IDL type

For CORBA-IDL there are two templates provided: `sequence` and `string`.

- There are two types of `sequences` :
  - bound, defined by a type and with a length known at compile time;
  - unbound, only defined by there type and a length that is known at run-time;
- There a two types of strings available:
  - bound, with fixed length known at compile time
  - unbound, known length at run-time.

IDL also provides arrays, that can be multi-dimensional but must have a fixed length at compile time.

By taking the cross-section of the MIDL and the OMG-IDL basic-types, IDL-files can be composed that use identical data-types and structures. This is very important for comparing the marshalling overhead for a specific data-type.

The IDL file that results, can also be cut into pieces. These pieces will only have the interface specification of the operations that will be conducted for a specific benchmark. This test can be done to check the influence of the size of the interface and the number of operations, and the thus resulting stubs.

The next table gives a cross-section on the tables of basic-types for OMG-IDL and MIDL.

These are the types that are used in the marshalling-benchmark.

Base Type	Description
boolean	8-bit data item
octet / byte	8-bit data item (no conversion)
char	8-bit data item
double	64-bit floating-point number
float	32-bit floating-point number
long	32-bit signed integer
short	16-bit signed integer

Next to these basic data-types, compound data-types will be defined like:

- bound and unbound arrays(or sequences);
- bound and unbound strings;
- structures, including all data-types mentioned above;
- arrays with structures and strings

Network data-size

The request-path in two-way calls are dependent on the network and the network-layer within the operating-system. The latencies that arise in those layers are very difficult to isolate and measure. By defining an IDL-file, with fixed array-lengths for every basic-type, the number of bytes transmitted will vary corresponding to the size of the basic-type. The number of transmitted bytes on the network will have a direct effect on the network-layers in the operating-system, which is a potential problem for comparing results. To eliminate this influence, we keep the network-load as a constant factor, independent of the transmitted basic-type.

This leads to the idea, to define a standard length where every type that must be transmitted, fits in one or more times.

The largest defined type to transmit, is the structure with all the common basic-types from the benchmark IDL. This leads to the following structure:

```
struct  {
        boolean,        // 1 byte
        char,           // 1 byte
        octet/byte,     // 1 byte
        double,         // 8 bytes
        float,          // 4 bytes
        long,           // 4 bytes
        short           // 2 bytes
        }
```

This gives a total of 21 bytes for the structure, which is not a nice number to work with and not a multiple of one the larger basic-types. This led to the insertion of three extra octets/bytes, so that the total number of bytes is 24, which is perfect for this purpose.

The structure can than be defined as:

```
struct  {
        boolean,        // 1 byte
        char,           // 1 byte
        octet/byte,     // 1 byte
        double,         // 8 bytes
        float,          // 4 bytes
        long,           // 4 bytes
        short           // 2 bytes
        octet/byte,     // 1 byte
        octet/byte,     // 1 byte
        octet/byte,     // 1 byte
        }
```

The amount of 24 bytes is perfect because nice sized arrays of basic-types can be created to get the same amount of network-data, as shown below.

```
double[3]       // 3 x 8 bytes   =   24 bytes
float[6]        // 6 x 4 bytes   =   24 bytes
long[6]         // 6 x 4 bytes   =   24 bytes
short[12]       // 12 x 2 bytes  =   24 bytes
octet/byte[24]  // 24 x 1 byte   =   24 bytes
char[24]        // 24 x 1 byte   =   24 bytes
boolean[24]     // 24 x 1 byte   =   24 bytes
```

The CORBA 2.0 IIOP specification ([OMG-97] chapter 12) presents the structure of IIOP-requests, that are strictly specified whereas this specification is the one the most important specifications for ORB interoperability. The specification also describes the alignment for data-members.

For both platforms, the alignment of the data on the network is equal to the alignment of the data in the computer-memory. This means that the basic-types are mapped on a multiple of their own size, with respect to the start of the data-segment in the network-packet. For a double this means that a new double-variable can only start on an address that is an multiple of 8, and for a long this address is an multiple of 4. The presented structure will not lead to 24 bytes of data on the network. Therefore, the members of the structure must be reshuffled, to the following structure:

```
struct  {
        double      // 8 bytes
        float       // 4 bytes
        long        // 4 bytes
        short       // 2 bytes
        boolean     // 1 byte
        char        // 1 byte
        octet/byte  // 1 byte
        octet/byte  // 1 byte
        octet/byte  // 1 byte
        octet/byte  // 1 byte
        }
```

For a more complete and detailed description of this alignment-phenomenon, see appendix B.

In this benchmark, several kinds of interfaces will be used. Also the size and the type of the data that is transported is varied.

For every type of invocation, a different interface will be defined, and they will have method calls like:

- void something (in)
- void something (in, out)

Increasing the network data

Increasing the size of the network data, can simple be done by changing the size of the arrays of parameters in the IDL file or on runtime for sequences. The sizes used during this benchmark are multiples of 24 (the size of the largest data-type, see previous section). The actual sizes measured are:

- 24 bytes
- 240 bytes
- 2,4 Kbytes
- 24 Kbytes
- 240 Kbytes

Results and evaluation

The results of all measurements are presented in appendix E. The matrices in this appendix, contain the mean, 5% lower-bound and 95% upper-bound limits of the response-times measured. This means that the intervals between the lower- and upper-bound represent the interval of 90% of all response times. This interval must be considered as an empirical 90% confidence interval.

The results presented in those matrices, show that DCOM is considerably faster in requests with small parameter-sizes. In the measurements with 24 or 240 byte parameters, DCOM is in average 2.5~3 times faster as its Orbix opponent. This effect is shown in Figure 15.

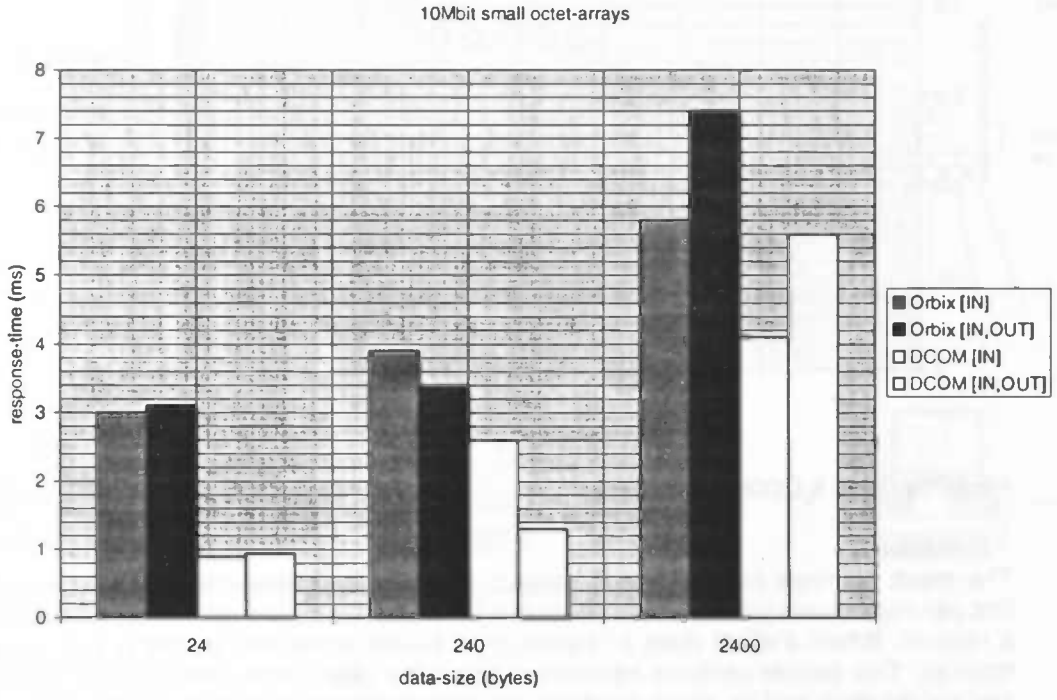


Figure 15:Orbix & DCOM, small octet-arrays

When the parameter-sizes increase, this speed-advantage effect decreases as the network becomes an eminent limiting factor. For the largest parameter-sizes, the response-times of DCOM and CORBA/Orbix are almost equal. This effect is shown in Figure 15.

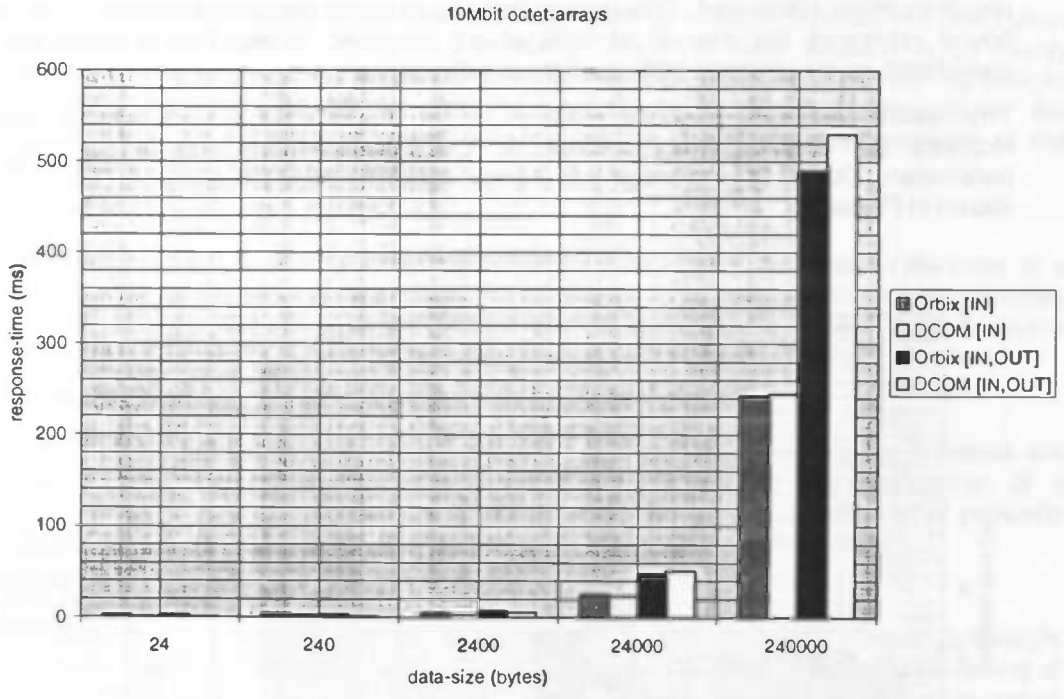


Figure 16:Orbix & DCOM, octet-arrays

The result matrices in appendix E, show that small parameter-sizes are almost equally fast per middleware product. This effect can be explained on the network overhead during a request. When a client does a request on a server, a request packet is sent over the network. This packet contains information about the client- and server-object, the client- and server-node and on which interface the request must be invoked. All this information requires a quite large network-packet on its own. Typically, such a request has a packet size of approx. 200 bytes, without any request parameters. For the network latency and transport-time, the packet doesn't get considerably larger when the parameter-size is pretty small, like 24 or 240 bytes. The same explanation holds for the return packet, in which the transport of a small array of return-values doesn't take considerable more time then a single return value.

conclusions:

- CORBA/Orbix has approximately 15~20% more marshalling/unmarshalling overhead for simple data-structures as defined for this experiment compared with basic IDL-types.
- CORBA/Orbix has approximately 30% more marshalling/unmarshalling overhead for octet-sequences. This can be explained by the fact that accessing a single byte on 32-bit processor is a relative expensive operation.
- DCOM has equal marshalling/unmarshalling overhead for all data-types.
- 'IN-OUT requests' are very efficient when used with small parameter-sizes. For large parameter-sizes the requests over an 'IN-OUT interface' require approximately twice the time for an 'IN interface' request.

Network configuration for Experiment F,H and L

The following figure gives the network configuration used during Experiment F,H,L. This configuration is also suitable for executing Experiments E,G and K. This picture reflects the office network of KPN Research in Groningen.

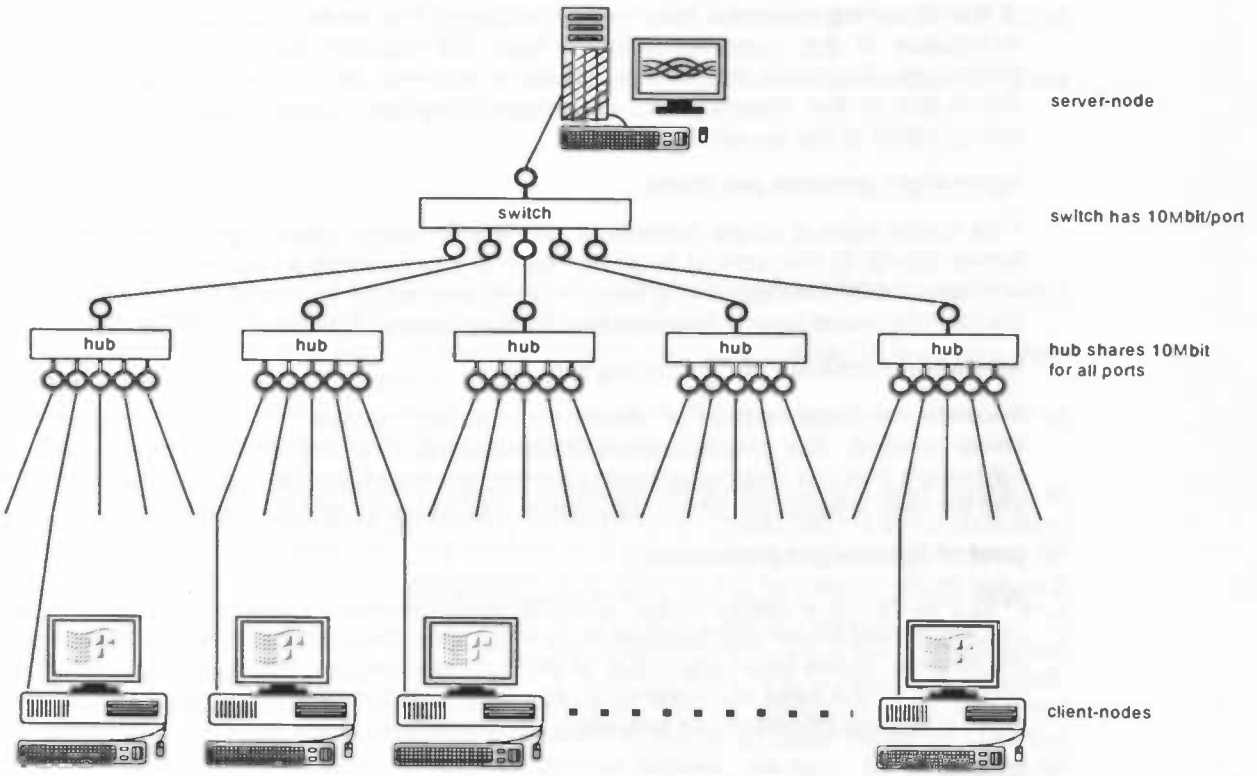


Figure 17:Multi client-node network-configuration

4.3.3 Experiment F

Implementation

The experiment description gives the following rules for developing an appropriate benchmark:

- the client-capsule has a single lightweight process.
- the request invoked on the server-object has a short parameter-list.
- the server-capsule has multiple server-objects / lightweight processes.

The first two requirements can be met very easy. The default type of capsule, is a capsule that has only one lightweight-process. As parameter used in the requests, a single 'long' is chosen. This means that the actual parameter-size on the network contains 4 bytes.

server-capsule with multiple objects / multiple threads

In this benchmark, different approaches are used for the server-capsules. Based on the technology the two platforms provide, the best mechanism for each platform is chosen to create a server-capsule that has multiple lightweight processes. The server-capsules that where developed for this benchmark were also used in the benchmarks of experiments H and L.

CORBA/Orbix

With the Orbix implementation of CORBA, there are four standard ways for making a server-capsule more responsive or to let it concurrently serve more requests:

- a lightweight process per server-objects
- a lightweight process per client
- a lightweight process per incoming request
- a pool of lightweight processes that serve the incoming requests on any server-object

The following paragraphs give a short overview on the different models that can be used to enable multiple lightweight processes in a server-capsule. For a more 'in depth' description of these models is refered to appendix C.

lightweight process per object

When multiple server-objects are used with their own lightweight process, the distribution of the incoming requests becomes a problem. The most ideal situation is an evenly distribution of the incoming requests over the objects and therefore the lightweight processes. Because the clients explicitly connect to a specific server-object, the distribution of the incoming requests depends on the clients, that are unaware of the configuration of the server-capsule.

lightweight process per client

This model uses a single lightweight process for every client that has a reference to a server-object. In the case of large numbers of clients, large amounts of server-resources are used, while the clients might not even invoke actual requests. Due to huge resource claims, this model seems inappropriate for situations with large numbers of clients.

lightweight process per incoming request

Because the large number of clients and the high repetition in which the requests are being invoked, this model seems inappropriate. The creation and destruction of a lightweight process causes operating system overhead, so the large number of requests and the high repetition in which they come, can cause significant overhead.

pool of lightweight processes

With this model, a fixed number of lightweight processes is created. These lightweight processes wait for incoming requests, to which they are assigned during the processing of the request. In this benchmark, it is no problem to have one object (which is programmed according to the rules of re-entrancy, ensuring mutual exclusion of shared data), and let several lightweight processes execute it simultaneously.

DCOM

For the server-capsule on DCOM, the most up-to-date technology is used in the form of the Microsoft Transaction Server (MTS). The MTS can be seen as a future standard facility in DCOM, for creating scaleable server-capsules. The MTS has a slight confusing, or maybe misleading, name, by indicating that it is explicitly developed for database transaction processing. The latter is absolutely not true, is provides a foundation for easy development of server-capsules that will be used by large numbers of clients. For a better overview of the functionality of the MTS, see appendix D.

Comparing the MTS with the pool-of-lightweight-processes

When we compare the mechanism that the MTS uses to make a server serve multiple requests in parallel, with the pool of lightweight processes of Orbix, the MTS uses an dynamically variable pool of lightweight processes instead of a fixed number. So these mechanism are most comparable, and when we vary the number of lightweight processes in Orbix during several runs, we can measure the overhead induced by the increasing number of context switches.

Due to a lack of available time, the only configuration for the Orbix server-capsule that is measured, is a configuration with 20 lightweight processes.

Results and evaluation

As this experiment serves as a measurement to find the lower-bound in response-times for these benchmark capsules while communicating over the KPN-Research office network as shown in appendix A. The results of this experiment can be found in the matrices in appendix F, but are also displayed in the following table:

CORBA/Orbix	3.9 ms
DCOM	1.6 ms

Like in experiment A, DCOM is a considerable faster then CORBA/Orbix. Both middleware products are 0.7 ms slower as measured in experiment A. The extra latency is most likely introduced by the Ethernet-switch.

Conclusions:

- an MTS based DCOM server gives quicker response times then a CORBA/Orbix implementation based on a pool-of-lightweight-processes.
- both middleware products experience an equal amount of delay through an extra component in the network configuration.

4.3.4 Experiment H

Implementation

To create a benchmark in which multiple client-capsules, running on different client-nodes start at the same time, a benchmark must be developed that:

- has a synchronisation-service on which the clients can synchronise for executing their invocation requests on the server-node.
- has clients that connect to a synchronisation-service for their synchronisation and to the server-node on which the invocation-requests are done.

To create a sustained load on the middleware induced by all the client-nodes involved in this experiment, the client capsules running on these nodes, must have a synchronisation mechanism. The client capsules must connect to a special server-object and request for the synchronisation. Because a client capsule must wait for his request to return before it can continue executing, it has to wait until the sync. server returns its call. The synchronisation server collects all the incoming requests from the client capsules and returns them at once. The moment of returning is determined by a configurable time-out. The time-out is started by the first client capsule that calls for synchronisation. The other client capsules have the time-out period to connect to the benchmark server and call the sync. server for synchronisation. When the time-out expires, the sync. server returns all synchronisation-requests and all client capsules will start invoking requests on the benchmark server. After the client capsule has done all its requests, it calls the sync. server again to notify that it is ready. When all client capsules are ready, the client capsules exit and a script sends the measurement results to the server-node for storage. [See appendix G for a more detailed description of the client-capsule and sync. server layout. This appendix also contains a few scripts that run on the client-nodes for starting the client-capsules and transferring the results back to the server-node.]

Results and evaluation

The results of this benchmark is also presented in appendix F, where it consists of the first column of both matrices. For convenience, the two columns of these matrices are given in the following matrix for better comparison.

# nodes	DCOM	Orbix
1	1.6	3.9
5	2.5	8
10	1.8	28
15	2	45
20	2.3	63
25	2.1	82
30	4.1	100
40	8.1	115
50	13	

When these results are put in a chart (Figure 18), they assume a linear relation between the number of client-nodes and the response-time. This assumption cannot be made because of the missing results for 35 and 45 client nodes. For CORBA/Orbix, there is a linearity between five and thirty client-nodes. But the result for forty client nodes does not

fit the linear relation. For DCOM, there are only four results that indicate a linear relation. These results are unusable for an extrapolation, which gives the response-time based on the number of client-nodes, when this number is larger then used in the experiment.

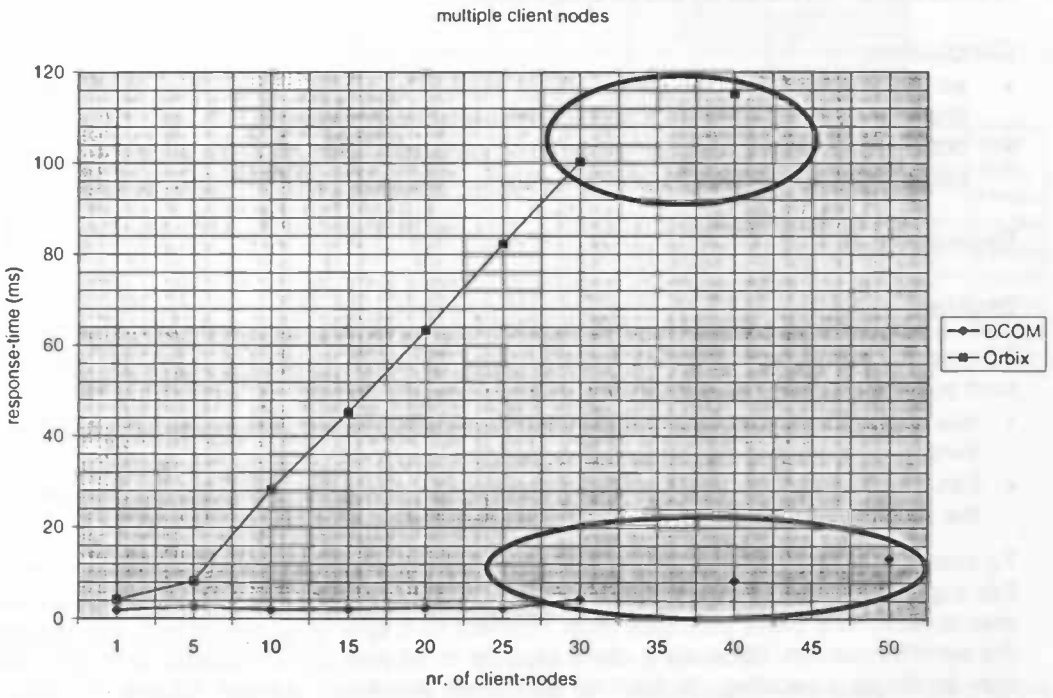


Figure 18:Orbix & DCOM, multiple nodes/response times

conclusions:

- response-times for DCOM and CORBA/Orbix increase when using multiple client-nodes
- both middleware products presume a linear relation within a certain interval
- response-times are considerably shorter for DCOM then for CORBA/Orbix
- the results are unusable for a extrapolation that predicts the response-time in case of larger number of client-nodes.

4.3.5 Experiment L

Implementation

This experiment, uses multiple lightweight processes per client-capsule, that invoke requests on the server-capsule. The client capsule developed for the previous benchmark must be extended to be use-able with multiple threads, while maintaining its synchronisation capabilities. To accomplish this, the client capsule must be extended with an internal synchronisation mechanism, which does the external synchronisation with the sync. server on behalf of all lightweight processes in the client capsule.

Results and evaluation

The results of this benchmark are presented in appendix F. The matrices in this appendix, contain the mean, 5% lower-bound and 95% upper-bound limits of the response-times measured. This means that the intervals between the lower- and upper-bound represent the interval of 90% of all response times. This interval must be considered as an empirical 90% confidence interval.

When examining Figure 19, it immediately shows the enormous difference in response-times between DCOM and CORBA/Orbix. Furthermore, each middleware implementation, shows a great linearity in its response times.

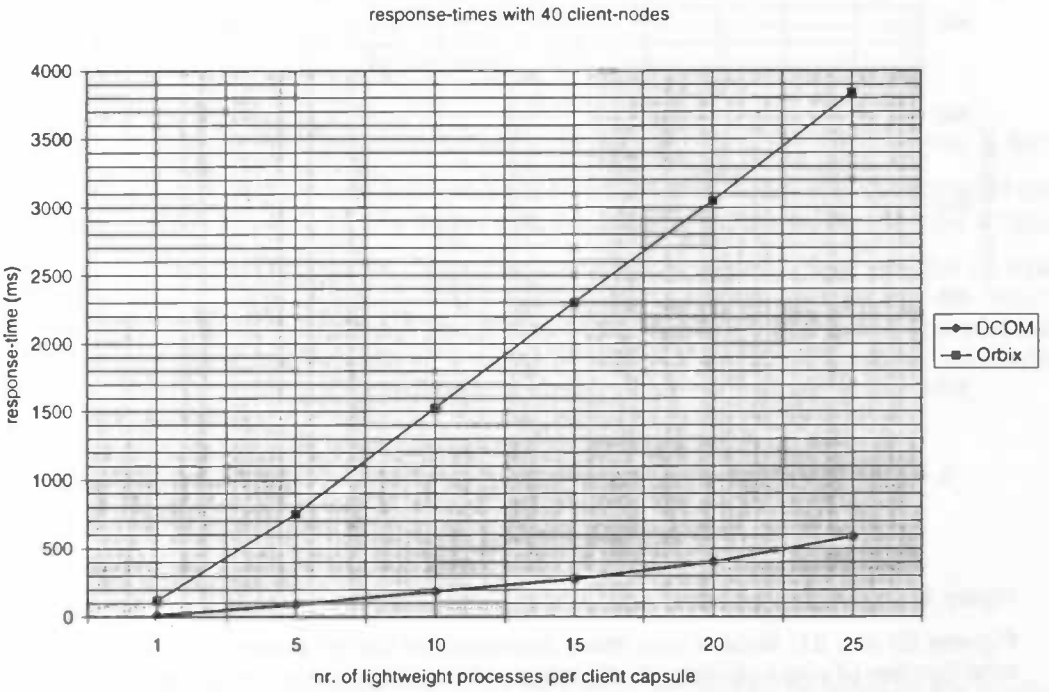


Figure 19:Orbix & DCOM, 40 client-nodes with multiple lightweight processes

Figure 20 and Figure 21 show all the results of the measurements with CORBA/Orbix and DCOM respectively in a separate graph.

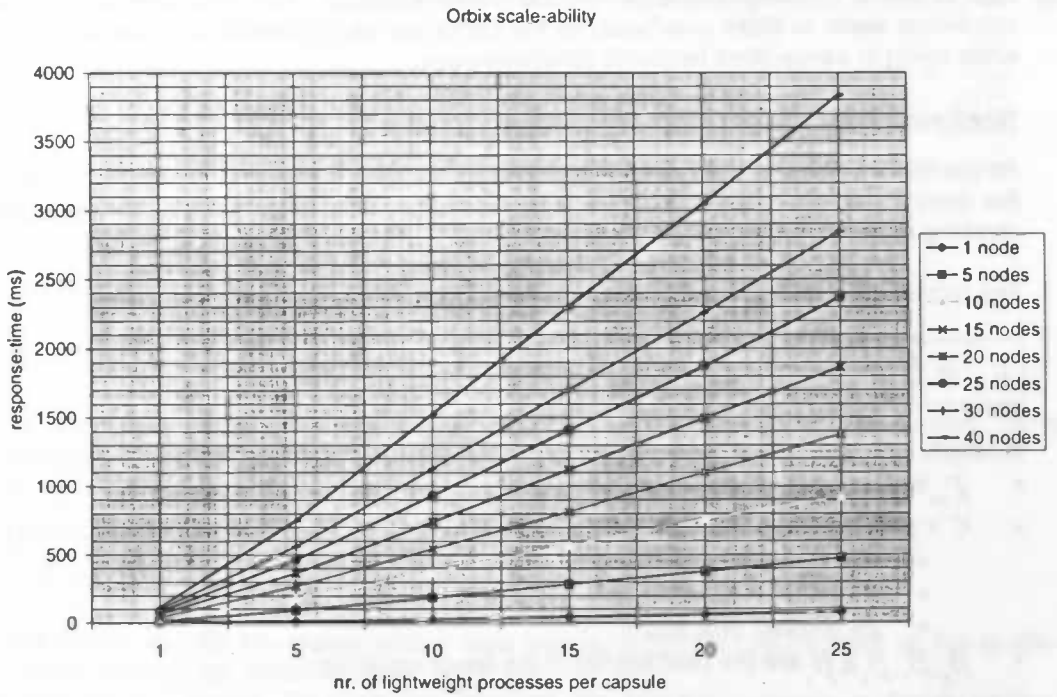


Figure 20:Orbix, scale-ability

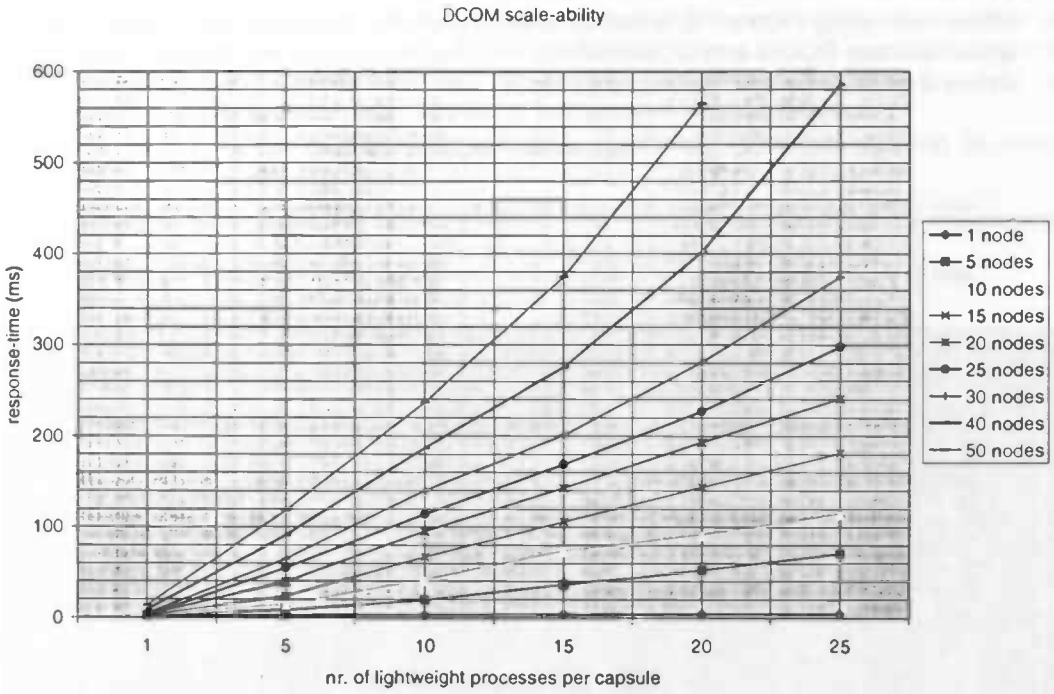


Figure 21:DCOM, scale-ability

Figures 20 and 21, show a nice linear behaviour of the response-time with respect to the total number of client-objects. While the graph for Orbix stays linear under high loads, the graph for DCOM shows a non-linear increase of the response-time. This can be explained by the overhead that is induced by the number of lightweight processes in the server-capsule. For Orbix, this number is fixated on twenty, while on DCOM, the MTS uses a dynamic number of lightweight-processes. During the benchmark-runs, this number of lightweight processes increased steadily from approx. five to one-hundred, as more client-objects were invoking requests on the server-capsule. This increase in lightweight processes leads to more overhead, so the performance of the server-capsule decreased while trying to serve more requests simultaneously.

Statistical analysis of scale-ability results

As the results show a high degree of linearity, multiple regression is used to investigate the linearity of the response times with respect to the total number of client-objects invoking requests on the server-capsule.

The model used for the linear multiple regression is:

$$T_{res} = \beta_0 + \beta_1 * X + \beta_2 * Y + \beta_3 * Z$$

in which:

- T_{res} is the dependent variable
- X, Y and Z are the independent variables
  - X= number of client-nodes
  - Y= number of lightweight-processes per client capsule
  - Z= product of X and Y
- $\beta_0, \beta_1, \beta_2 \& \beta_3$ are the coefficients of the linear equation.

This model was chosen due to highly dependence of the response-time with respect to the total number of client-objects, simultaneously invoking request on the server-node. The analysis is executed with the statistical program SPSS 7.0, from which the output is included in appendix G.

If we summarise the results, we see for CORBA/Orbix, the following coefficients:

β_0 = -4.6766

β_1 = -0.4358

β_2 = -1.0209

β_3 = 3.8487

Although this model fits the measurement data very well, the constant β_0 has a 95% confidence interval that includes zero (see appendix G). This means that this constant can have any value in the interval, but it may also be zero. Furthermore, the β_0 has a value under zero, which makes sense. This situation is induced by the fact that the multiple regression tries to fit the model to most linear part of the results. In this case, the results for the measurements with a small number of client-objects, lie in the non-linear part of the data. To analyse the influence of a model without a constant, we used following non-linear model, to investigate the change in parameters and the increase of the error.

$$Tres = \beta_1 * X + \beta_2 * Y + \beta_3 * Z$$

The coefficients are now:

β_0 = -0.6119

β_1 = -1.3039

β_2 = 3.8549

This model also fits the measurement data very good, which can be seen on the quality-specifier R-squared, which is nearly 1.0. A value of 1.0 for R-squared, means that the predicted values of the model are almost exactly the same as the results obtained with the measurements. This can be explained, on the fact that the constant β_0, has a small value, which influence is very small compared the other coefficients. Based on this model, the response time for this CORBA/Orbix environment (hardware and software), can be predicted with the following formula:

$$Tres = -0.6119 * nr\_pc - 1.3039 * nr\_lw\_pr + 3.8549 * (nr\_pc * nr\_lw\_pr)$$

For DCOM the same linear model is used, which resulted in the following coefficients:

β_0 = -0.8282

β_1 = -0.4688

β_2 = -0.7129

β_3 = 0.5562

For DCOM, the constant also has a 95% confidence interval, which includes zero. Using the same non-linear model without the constant, the following coefficients are obtained:

β_0 = -0.4945

β_1 = -0.7640

β_2 = 0.5579

This model also fits the measurement data very good, which can be seen on the quality-specifier R-squared, which is nearly 1.0.
Based on this model, the response time for this DCOM environment (hardware and software), can be predicted with the following formula:

$$Tres = -0.4945 * nr\_pc - 0.7640 * nr\_lw\_pr + 0.5579 * (nr\_pc * nr\_lw\_pr)$$

Analysis of the network-load

During the execution of the benchmarks, the network performance was monitored, as this is a potential bottleneck. The network-load is measured by examining the number of incoming packets on the server-node. These packets all have a size of approximately 200 bytes. Based on the packet-size and the number of packets/second, the limit of 10Mbit/second of the network has not been a bottleneck for the experiment.
According to the network-administrator of KPN Research, "this was nothing to worry about".

The following picture gives the network-load during one day of executing measurements on Orbix:

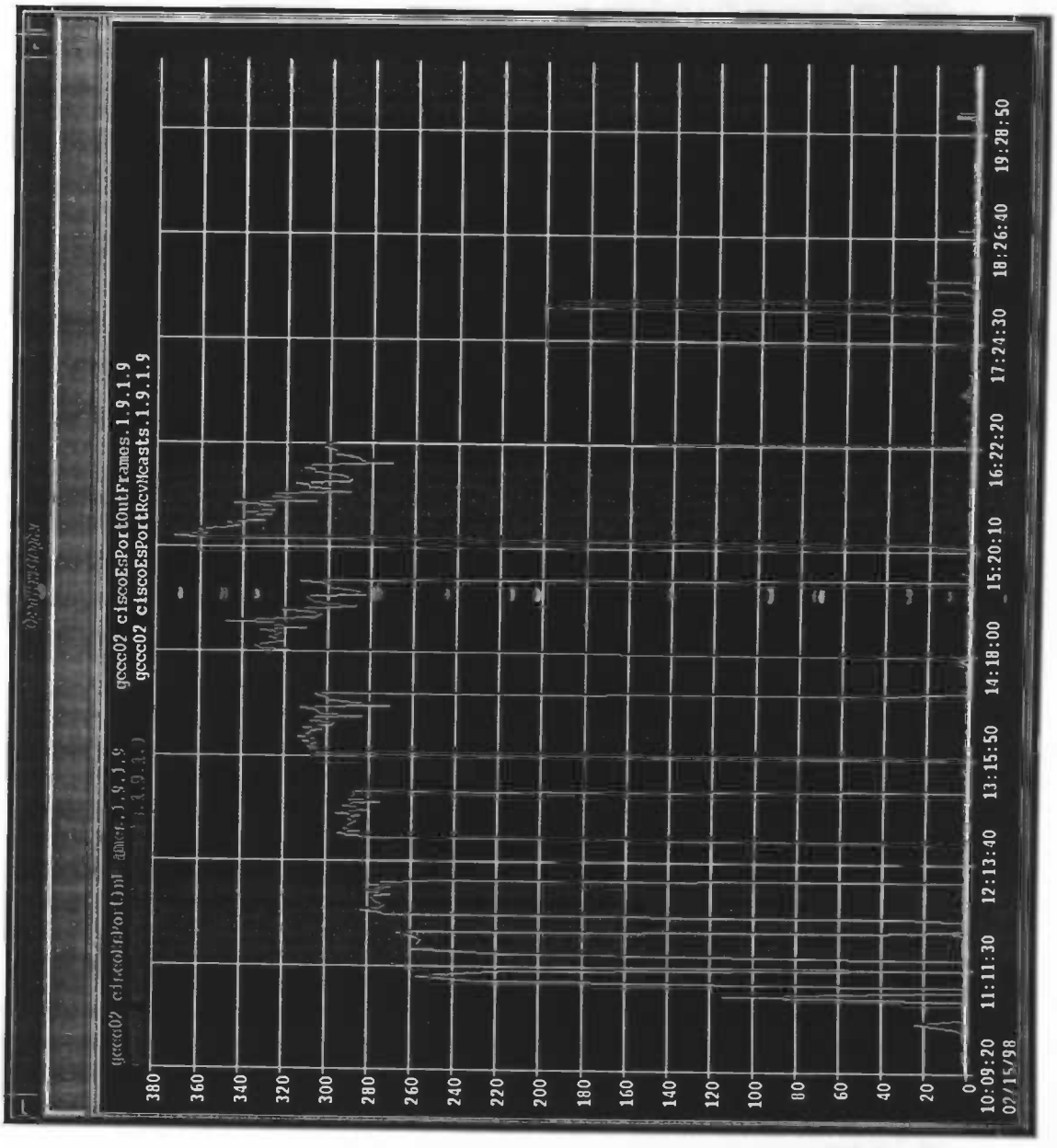
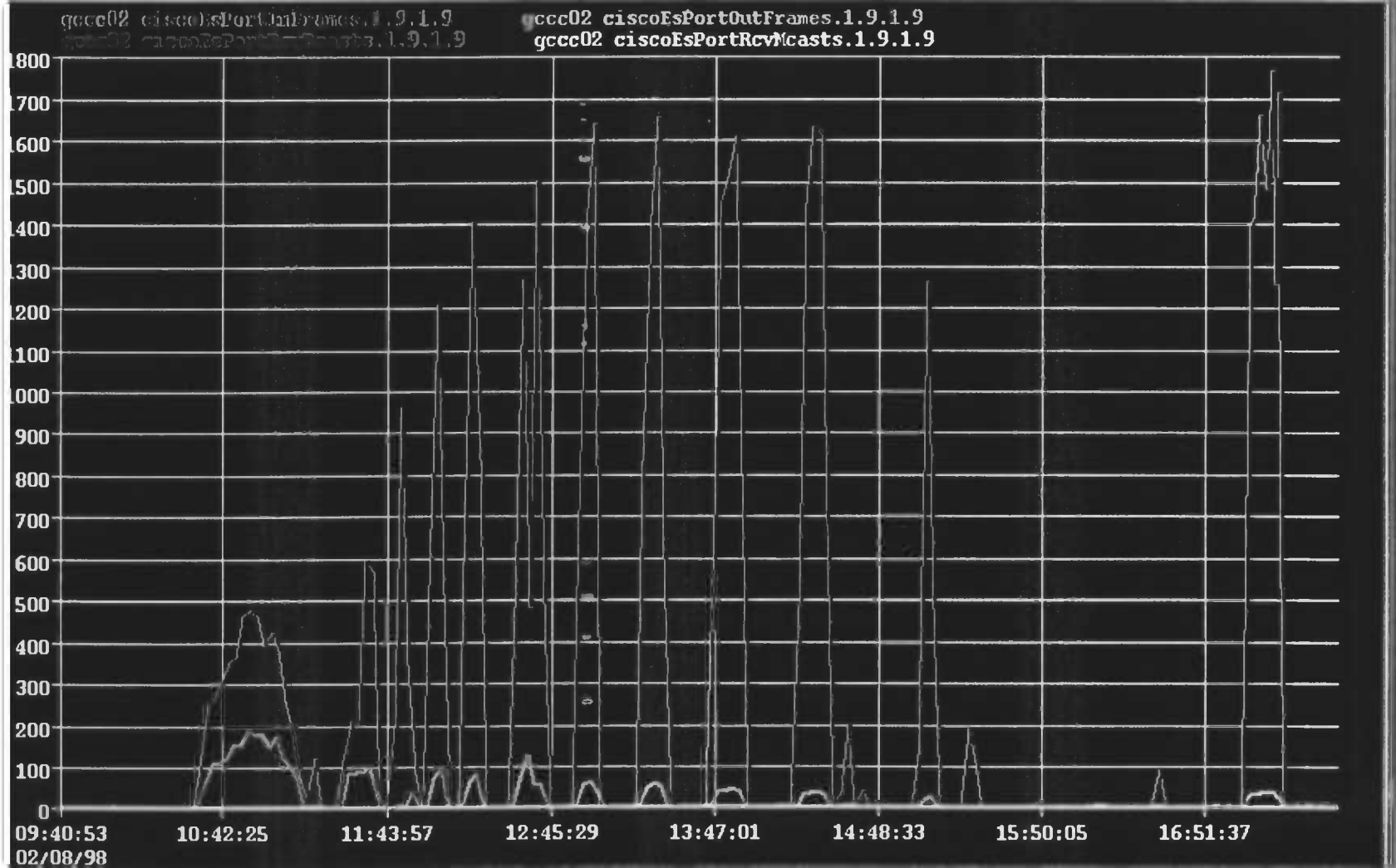


Figure 22:Orbix network-load

When looking at this graph, we see an repeating pattern in the number of incoming packets at the server-node. The graph shows the execution-time of the benchmarks very clearly. During every benchmark-run, the number of client-nodes was kept constant, while the number of client-objects on the client-nodes increased. As can be seen in the graph, every benchmark-run shows a decrease in the number of incoming packets, as the number of client-objects per client-node increases. This is visualised in Figure 24.


gccc02 ciscoEsPortOutFrames.1.9.1.9
gccc02 ciscoEsPortRcvMcasts.1.9.1.9
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
09:40:53
02/08/98
10:42:25
11:43:57
12:45:29
13:47:01
14:48:33
15:50:05
16:51:37

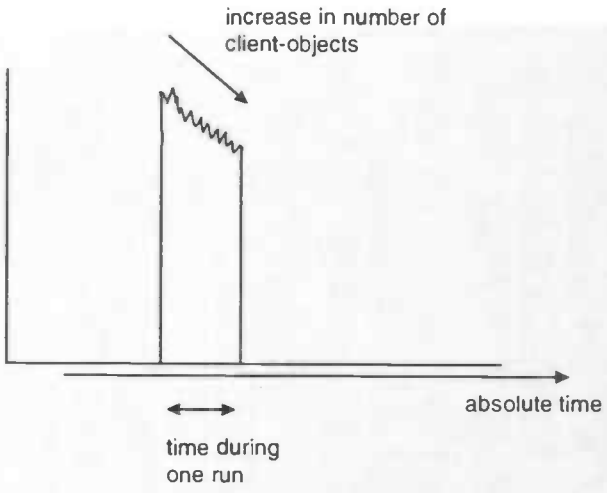


Figure 24:Network-load graph explanation

When observing these network-monitoring graphs, a clear upper-bound in the number of incoming packets on the server-node can be identified. As every client-object invokes a new request as soon as the previous request returns, the number of incoming packets can be seen as an indication for the number of requests a specific middleware implementation can serve per second. Based on these graphs, not only the number of served requests can measured, but the increase of the overhead on the server-node can also be found by examining the decrease of the number of incoming packets. This observation leads to the idea that there is direct relation between the response-time, the total number of running client-objects and the number of incoming packets. Roughly said, the response-time can be expressed by the total number of client-objects divided by the number of incoming-packets.

conclusions:

- the response-time that is experienced by a client-object, increases linear with the total number of client-objects.
- the response-time that is experienced by a client-object, is largely dependent on the total number of client-objects invoking requests.
- the response-time that is experienced by a client-object, is largely independent of the number of client-nodes and the number of simultaneously running client-objects on these nodes.
- when using requests with small parameter-sizes, the network-load is not a bottleneck.
- DCOM is 400 to 800% faster than Orbix.
- the variable number of lightweight processes of the MTS create a measurable overhead, as the number of lightweight processes increase when a large number of client-objects are invoking requests.
- the number of requests that a middleware implementation can serve per second can be found by examining the number of incoming packets on the server-node.
- an increase of the number of lightweight processes can give a decrease in the performance of a server-capsule when this number becomes to large. This effect can be seen by the server-capsule running in the MTS.

4.3.6 Result overview

This section gives an overview of the results and conclusion, which are based on the results obtained by the execution of the benchmarks of experiment A,F,H and L.

- the marshalling and un-marshalling process is largely independent of the used data-types. Only complex data-structures will increase the overhead here.
- the response-time of requests with small parameter-sizes is not significantly longer then parameter-less requests, due to the large header which is required for the request.
- requests with small return-parameter sizes do not significantly increase the response-time over requests without return-values.
- the response-time of requests with large parameter-sizes are largely dependent on the network latency.
- the response-time a client-object experiences is largely dependent on the total number of client-objects that simultaneously invoke requests.
- the response-time increases linearly with the increase of the number of client-objects that simultaneously invoke requests.
- DCOM has a clear performance advantage over CORBA/Orbix.
- the behaviour of CORBA/Orbix shows less variation in response-times under heavy load then DCOM. This effect can be seen by observing the confidence intervals of the measurement results.
- using requests with small parameter sizes, does not create excessive network-load with large numbers of clients.
- the MTS that is used for the server-capsule in the DCOM experiments doesn't cause measurable overhead.
- the MTS shows a non-linear behaviour under high load, due to the overhead induced by the large number of lightweight processes it uses.
- both server-capsules showed a linear increase in the required server-resources.
- the server-capsules claim all the CPU-resources when exposed to a sustained load of requests, invoked by large numbers of client-objects.

4.4 Summary

In this chapter, several experiments described possible benchmarks to evaluate the behaviour of middleware products under various conditions. Some of these benchmarks are especially meant to evaluate the behaviour of a specific component in the middleware architecture, while other benchmarks are meant to evaluate the behaviour under a heavy and sustained workload.

Based on the results of the benchmarks that were developed for DCOM and CORBA/Orbix, it can be said that the experiments expose the behaviour if the middleware clearly. The measurement results show, that the behaviour of the evaluated middleware products shows a linear relation with the number of client-objects that simultaneously invoke requests.

The measurement show furthermore that DCOM is considerably faster than CORBA/Orbix. Especially requests with small parameter-sizes and under heavy workload induced by many client-objects, the performance difference ranges from 400 to 800 percent.

5 Conclusions & Recommendations

Conclusions:

This report describes a comparative performance evaluation of DCOM and the CORBA implementation Orbix from Iona Technologies. To get a good overview of both middleware infrastructures, their internal architecture was evaluated and the functionality of the identifiable components described. This evaluation led to a generic architecture for middleware, which provides the components and functions of both evaluated architectures. With the knowledge of the generic architecture in mind, and focusing on measuring the performance and overhead of the components in the architecture, the most suitable performance evaluation technique had to be determined. The synthesised benchmarking technique was the most suitable technique for this performance evaluation. A set of experiments has been developed to measure the overhead or performance behaviour of the components in the generic architecture. A subset of these experiments have been implemented and executed on both middleware implementations, which resulted in a large amount of measurement data. Based on the analysis of the data, some conclusions on the performance and scale-ability of these middleware implementations are drawn.

For the evaluation of the middleware architectures, the following conclusions apply:

- The architectures of CORBA and DCOM are very alike, when looking at the functionality of the individual components that can be identified in the architectures.
- Based on the similarity of both architectures, a generic middleware architecture is designed, which fits both CORBA and DCOM.
- As both CORBA and DCOM are evolving, and more products become available, both middleware infrastructures become more useful for Internet and back-office applications.
- The higher level of abstraction and the demand for interoperability in CORBA is reflected in the network-protocol used for transporting request-messages. Where DCOM uses pointer-fables and transmits pointer indirection's for an particular operation on an object, CORBA transmits the names of the object, interface and operation.
- Based on the observation of the network-protocols, the comparison of the names in CORBA is expected to use more processor-time than the pointer-indirection's in DCOM.
- DCOM can be seen as a networked extension of the inter-process communication (IPC) infrastructure used in 32bits operating systems of Microsoft, like Windows NT4.0. It is therefore very likely that this IPC infrastructure is highly optimised for the use in the operating systems, without being prepared to offer interoperability with other platforms.
- When using CORBA implementations on Microsoft Windows NT, they are always supplied by 3rd party vendors and running on top of the operating system instead of being incorporated with it. This will give more overhead as an additional interface layer to the operating system must be used, while it is probably less optimised for Windows NT as DCOM.

Based on the evaluation of the available performance evaluation techniques, and the search for the most suitable technique for this performance evaluation, the following conclusions apply:

- Based on the complexity of distributed systems and the use of commercially available middleware implementations, measurements are the best feasible performance evaluation method.
- For measuring the influence or behaviour of the identifiable components in the generic architecture, the synthesised benchmark is the most suitable technique. This type of benchmark allows the flexibility needed for generating high loads on the individual components in the generic architecture.

Based on the development of the experiments and the results obtained during the execution of the benchmarks, the following conclusions apply:

- The set of experiments that is developed for the performance evaluation, is suitable for determining the performance and scale-ability of middleware.
- While a total of ten experiments where developed, the results of the scale-ability experiments should be analysed as a whole, for evaluating the behaviour of the middleware when increasing the load on or changing the configuration of the server-capsule.
- Marshalling is nothing more than writing all the parameters in a flattened buffer of bytes, according the network data-format that is used.
- A sender, always transmits its data in its native byte-order. The receiving side must check for the correspondence in byte-order and is responsible for necessary conversions.
- The alignment of data-members of a request transported over a network is equal to the alignment in main-memory. As data-types are aligned on memory boundaries that are a multiple of their own size, in many cases gap-fillers must be used. This can cause considerably more network traffic then actually needed for transporting the request.
- Based on the results obtained in benchmarks that cover the marshalling of the requests, it can be said, that the marshalling is largely independent of the data-type that is used as parameter in the request. Only complex data-types, like structures cause more overhead then basic IDL-types.
- Requests and replies between a client and a server always contain a fairly large header of approx. 150~200 bytes. Request or replies with small parameter-sizes (like a 50~100 bytes) are therefore almost as fast as requests without any parameters.
- As the parameter-sizes of the requests increase, the performance becomes limited to the network-throughput and the network-latency.
- The results of the scale-ability measurements show less variation in response-times of Orbix under high loads then DCOM. This effect can be seen by observing the confidence intervals of the measurement results. This effect is most likely caused by the dynamic creation and deletion of lightweight processes in the MTS, which causes overhead on irregular intervals.
- Using requests with small parameter sizes, does not create excessive network-load with large numbers of clients. The network-load is determined by the number of packets on the network, which is bound to the performance of the server-node. Every request which is served by the server-capsule, results in a new request from the just served client-object. This means that network-load is determined by the number of requests that the server-capsule serves per second.
- The MTS that is used for the server-capsule in the DCOM experiments doesn't cause measurable overhead, compared to a server-capsule not running inside the MTS.
- Orbix's pool of lightweight processes which is used as solution for the server-capsule in the scale-ability benchmarks, shows a perfectly linear behaviour under high loads.
- The MTS shows a non-linear behaviour under high load, due to the overhead induced by the increasing number of lightweight processes that it creates.
- Both server-capsules showed a linear increase in the required server-resources, like main-memory or total execution-time.
- In the area where the scale-ability measurements show a linear increase of the response-times, DCOM has response-times which are a factor four to eight faster than the response-times for Orbix.

- The linear increase of the response-times can be explained by the fact that the server-capsule is fully saturated. This means, that there is no processor-time left for serving additional requests, which forces the server-capsule to put the incoming requests in a queue until they can be served.

During the development of the benchmarks, a lot of hands-on experience was gained. With respect to the development of applications with CORBA and DCOM, the following conclusions apply:

- The development of DCOM server-objects without the wizard in Microsoft Visual C++ 5.0 is very troublesome. In older versions of the Microsoft development environment, there is no active support for DCOM objects. This makes the development of DCOM objects very difficult. Version 5.0 has a wizard for the creation of the correct IDL-files, the configuration of the development environment, creation of an DLL for the proxy and stub code and the correct registry entries for the object.
- The development tools for the used CORBA implementation are pretty good. Orbix is supplied with a number of small tools for registering the server-objects and manipulating these entries. These entries are based on configuration files instead of the registry of the operating-system.
- Using the MTS for creating server-capsules that handle large numbers of clients requires no special effort of the programmer. No extra programming is required to protect shared state of concurrent objects

Recommendations:

Although a thorough evaluation of the middleware architectures has been done, and a subset of the experiments have showed the performance and scale-ability of the middleware implementations, there are a number of recommendations and suggestions for future research.

These recommendations and suggestions are partially based on practical execution of the experiments, as well as further research for a more detailed generic architecture or the development of new experiments:

- Due to fact that only a subset of the experiments where implemented and executed, there are experiments that are not executed, that may provide important information on the performance and behaviour of certain components in the architecture. These experiments, therefore should be implemented and executed.
- To obtain results with a larger number of client-objects invoking requests, the number of client-nodes could be increased. This can provide information on the limitations of the middleware implementations and the validity of the model used in the statistical analysis. The non-linear behaviour of the MTS may indicate that it needs another model, which has an extra contribution in the response time, which depends on the number of lightweight processes used.
- In this comparative performance evaluation, there where only two middleware implementations evaluated. CORBA has many implementations, which makes it very interesting to conduct measurements on different implementations for determining their performance and scale-ability.
- As the acquired results yield over eight million numbers (measured response-times), and time to analyse them was very limited, this data may contain more information than is presented in this report. Therefore, it can be worthwhile to analyse the results on further relations.
- For a better comparison of DCOM and Orbix, the experiments on Orbix could be conducted with different server-capsule configurations, to examine the influence of the degree of parallelism in the server-capsule.
- With the release of DCOM for other operating systems than Microsoft Windows platforms, it is very interesting to evaluate the performance of DCOM on non-Microsoft operating systems.
- To investigate the performance of the middleware implementations when requests are made between nodes with different processor architectures, the experiments should be executing on heterogeneous hardware platforms and operating systems.

- Executing the experiments on networks with different latencies or throughput. The experiments where executed on a single network type. By using networks with different network-latencies, the influence of the network-latency on the total response-time can determined. By using networks with other throughputs, the performance of the middleware can become an determining factor for the response time for requests with large parameter-lists.

6 References

DCOM

[EDDON-98] *Understanding the DCOM Wire Protocol by Analyzing Network Data Packets*
Guy Eddon & Henry Eddon
Microsoft Journal March 1998/vol.13/no.3

[BOX-98B] *Q&A ActiveX/COM*
Don Box
Microsoft Journal March 1998/vol.13/no.3

[PLATT-98] *Give ActiveX-based Web Pages a Boost with the Apartment Threading Model*
David Platt
Microsoft Journal February 1998/vol.12/no.2

[SHEPHERD-98] *The Visual Programmer*
George Shepherd & Scot Wingo
Microsoft Journal February 1998/vol.12/no.2

[BOX-98A] *Q&A ActiveX/COM*
Don Box
Microsoft Journal January 1998/vol.11/no.1

[CHAPPELL-98] *How Microsoft Transaction Server Changes the COM Programming Model*
David Chapell
Microsoft Journal January 1998/vol.11/no.1

[BROWN-98] *Distributed Component Object Model Protocol – DCOM/1.0*
Nat Brown & Charlie Kindel
Microsoft (published as an IETF Draft) 1998

[GRIMES-97] Professional DCOM Programming
Richard Grimes
Wrox Press 1997

[HORST-97] *DCOM Architecture*
Markus Horstmann & Mary Kirtland
Microsoft WhitePaper

[REED-97] *Microsoft Transaction Server Helps You Write Scalable, Distributed Internet Apps*
Dave Reed, Tracey Trewin & Mai-lan Tomsen
Microsoft Journal August 1997/vol.12/no.8

[PLATT-97] *Fashionable App Designers Agree: The Free Threading Model is What's Hot This Fall*
David Platt
Microsoft Journal August 1997/vol.12/no.8

[GOSWELL-95] *The COM Programmer's Cookbook*

Crispin Goswell
http://www.microsoft.com/oledev/olecom/com_co.htm

[BROCK-95] *Inside OLE, second edition*
Kraig Brockschmidt
Microsoft Press, 1995

[MICROS-96A] *Windows NT Server, DCOM, A Business Overview*
Microsoft, 1996

[MICROS-96B] *Windows NT server, DCOM Technical Overview*
Microsoft, 1996

[BROCK-96] *What OLE Is Really About*
Kraig Brockschmidt, July 1996
http://www.microsoft.com/oledev/olecom/aboutole.htm

[MICROS-94] *The Microsoft Object Technology Strategy*
June 1994
http://www.microsoft.com

CORBA

[ORFALI-97B] *Instant CORBA*
Robert Orfali, Dan Harkey & Jeri Edwards
John Wiley & Sons, inc 1997

[ORFALI-97A] *Client/Server Programming with JAVA and CORBA*
Robert Orfali & Dan Harkey
John Wiley & Sons, inc 1997

[OMG-97] *The Common Object Request Broker Architecture and Specification, Revision 2.0*
OMG document 97.2.25
1997

[IONA-97C] *Orbix Programmer's Guide (ver. 2.3)*
Iona Technologies
October 1997

[IONA-97B] *Orbix Programmer's Reference (ver. 2.3)*
Iona Technologies
October 1997

[GOKHALE-97A] *Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance*
Gokhale & Schmidt, 1997
hicss-97.ps on http://siesta.cs.wustl.edu/~schmidt

[GOKHALE-97B] *Evaluating CORBA latency and Scalability Over High-Speed ATM Networks*
Gokhale & Schmidt, 1997
icdcs-97.ps on http://siesta.cs.wustl.edu/~schmidt

[SCHMIDT-97] *Overview of the TAO Architecture*
Schmidt, 1997
http://siesta.cs.wustl.edu/~schmidt/TAO-architecture.htm

[IONA-96] *The Orbix Architecture*
IONA Technologies, November 1996
Arch_a4.doc v1.4 on http://www.iona.com

Performance evaluation

[KANT-92] *Introduction to computer system performance evaluation*
K.Kant
McGraw-Hill 1992

[WAL-90] *Computer prestatie analyse*
S.G. van der Wal
Academic Service 1990

Miscellaneous

[ACZEL-92] *Complete business statistics (second edition)*
Amir D. Aczel
Irwin 1992

[KATIYAR-97] *Notes on OLE and CORBA*
Dinesh Katiyar
http://theory.stanford.edu/pub/katiyar/misc/ole_vs_corba.html

[RAYMOND-97] *Reference Model of Open Distributed Processing (RM-ODP): Introduction*
Kerry Raymond
University of Queensland

[ISGINC-97] *Middleware White Paper*
International Systems Group Inc.
http://www.openvms.digital.com/openvms/whitepapers/middleware/isgmidware.html

[PUDER-97] *Verteilte Objekte: DCOM versus CORBA*
Arno Puder
iX-magazin 1997

[CHUNG-97] *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer*
Chung, Huang & Yajnik, 1997
http://akpublic.research.att.com/~ymwang/papers/HTML/DCOMnCORBA/S.html

[ROY-96] *Choosing between CORBA and DCOM*
Roy & Ewald
Object Magazine, Oktober 1996

[ORFALI-96] *The Essential Distributed Objects Survival Guide*
Orfali, Harkey & Edwards
1996

[PURE-97] *Quantify User's Guide*
Pure Software Inc. 1997
http://www.pureatria.com

```
// benchmark IDL
// version 1.1
// Richard Beekhuis at KPN Research Nov.1997
//
// DCOM IDL file

// This IDL-file contains the following interfaces for the benchmark:
//          void test(in)
//          void test(inout)
//

#ifndef DO_NO_IMPORTS
import "unknwn.idl";
import "oaidl.idl";
#endif

// Structure with all basic types:
// This structure is especially created with a length of 24 bytes, with all the arguments on their
// natural alignment boundaries
typedef struct _BasicTypes
{
        double  DoubleVal;      // 8 bytes
        float   FloatVal;       // 4 bytes
        long    LongVal;        // 4 bytes
        short   ShortVal;       // 2 bytes
        boolean BooleanVal;     // 1 byte
        char    CharVal;// 1 byte
        byte    OctetVal1;      // 1 byte
        byte    OctetVal2;      // 1 byte
        byte    OctetVal3;      // 1 byte
        byte    OctetVal4;      // 1 byte
                                // ---------------- +
                                // total: 24 bytes
} tBasicTypes;



// the index '_24*' means that the resulting on the network will be this size in bytes
// the array index gives the number of data-items to achive this

// Arrays of octets / bytes
typedef byte            tOctet_24[24];
typedef byte            tOctet_240[240];
typedef byte            tOctet_2K4[2400];
typedef byte            tOctet_24K[24000];
typedef byte            tOctet_240K[240000];

// Arrays of doubles
typedef double          tDouble_24[3];
typedef double          tDouble_240[30];
typedef double          tDouble_2K4[300];
```

```
typedef double          tDouble_24K[3000];
typedef double          tDouble_240K[30000];

// Arrays of compound types:
typedef tBasicTypes     tBasicTypes_24[1];
typedef tBasicTypes     tBasicTypes_240[10];
typedef tBasicTypes     tBasicTypes_2K4[100];
typedef tBasicTypes     tBasicTypes_24K[1000];
typedef tBasicTypes     tBasicTypes_240K[10000];

// Arrays of template types:
typedef char            *tString_24[24];
typedef char            *tString_240[240];
typedef char            *tString_2K4[2400];
typedef char            *tString_24K[24000];
typedef char            *tString_240K[240000];

// ************************************************************************
// ************************************************************************

// Sequences of basic types(UNBOUND):
typedef byte            tOctetSeq[*];
typedef double          tDoubleSeq[*];

// Sequences of template types(UNBOUND):
typedef char*   tStringSeq;

// Sequences of compound types(UNBOUND):
typedef tBasicTypes     tBasicTypesSeq[*];


// The include files contain the actual interface-descriptions
#include "twoway_in.idl"
#include "twoway_inout.idl"

[
   uuid(B5A43981-6659-11d1-9142-00805F84F1FD),
   helpstring("Marshaling Benchmark Type Library"),
   version(1.0)
]

library BenchMarkLib
{
   importlib ("stdole2.tlb");

   // benchmark class
   [
   // For insertion in client en server code, used for registering and instantiating the object
   // {A32C3EFD-64C4-11d1-9141-00805F84F1FD}
   // DEFINE_GUID(CLSID_Benchmark, 0xa32c3efd, 0x64c4, 0x11d1, 0x91, 0x41, 0x0, 0x80, 0x5f,
0x84, 0xf1, 0xfd);
   //

         uuid(A32C3EFD-64C4-11d1-9141-00805F84F1FD),
      helpstring("Benchmark Class")
   ]

   coclass Benchmark
   {
      [default]   interface ITwoway_in;
                  interface ITwoway_inout;
   }
}
```

The following file shows the interface specification for the IN-interface:

```
[
        uuid(A32C3EF0-64C4-11d1-9141-00805F84F1FD),
```

```
        object
]

interface ITwoway_in : IUnknown
{
// No Parameters
        HRESULT NoParameterReq();
// Sequences
        // Sequences of basic types
        HRESULT OctetSeq        ([in] long nSize, [in, size_is(nSize)] tOctetSeq   OctetSeqIn);
        HRESULT DoubleSeq       ([in] long nSize, [in, size_is(nSize)] tDoubleSeq  DoubleSeqIn);
        // Sequences of template types
        HRESULT StringSeq       ([in, string] tStringSeq  StringSeqIn);
        // Sequences of compound types
        HRESULT BasicTypesSeq   ([in] long nSize, [in, size_is(nSize)] tBasicTypesSeq BasicTypesSeqIn);
// Arrays
        HRESULT Octet_24   ([in] tOctet_24    Octet_24In);
        HRESULT Octet_240  ([in] tOctet_240   Octet_240In);
        HRESULT Octet_2K4  ([in] tOctet_2K4   Octet_2K4In);
        HRESULT Octet_24K  ([in] tOctet_24K   Octet_24KIn);
        HRESULT Octet_240K ([in] tOctet_240K  Octet_240KIn);

        HRESULT Double_24   ([in] tDouble_24    Double_24In);
        HRESULT Double_240  ([in] tDouble_240   Double_240In);
        HRESULT Double_2K4  ([in] tDouble_2K4   Double_2K4In);
        HRESULT Double_24K  ([in] tDouble_24K   Double_24KIn);
        HRESULT Double_240K ([in] tDouble_240K  Double_240KIn);

        HRESULT String_24   ([in] tString_24    String_24In);
        HRESULT String_240  ([in] tString_240   String_240In);
        HRESULT String_2K4  ([in] tString_2K4   String_2K4In);
        HRESULT String_24K  ([in] tString_24K   String_24KIn);
        HRESULT String_240K ([in] tString_240K  String_240KIn);

        HRESULT BasicTypes_24   ([in] tBasicTypes_24    BasicTypes_24In);
        HRESULT BasicTypes_240  ([in] tBasicTypes_240   BasicTypes_240In);
        HRESULT BasicTypes_2K4  ([in] tBasicTypes_2K4   BasicTypes_2K4In);
        HRESULT BasicTypes_24K  ([in] tBasicTypes_24K   BasicTypes_24KIn);
        HRESULT BasicTypes_240K ([in] tBasicTypes_240K  BasicTypes_240KIn);
};
```

The following file gives the specification the IN,OUT-interface:

```
[
        uuid(A32C3EFC-64C4-11d1-9141-00805F84F1FD),
        object
]

interface ITwoway_inout : IUnknown
{
// Sequences
        // Sequences of basic types
        HRESULT OctetSeq        ([in,out] long *nSize, [in,out, size_is(*nSize)] tOctetSeq   OctetSeqIn);
        HRESULT DoubleSeq       ([in,out] long *nSize, [in,out, size_is(*nSize)] tDoubleSeq
DoubleSeqIn);
        // Sequences of template types
        HRESULT StringSeq       ([in,out, string] tStringSeq  StringSeqIn);
        // Sequences of compound types
        HRESULT BasicTypesSeq   ([in,out] long *nSize, [in,out, size_is(*nSize)] tBasicTypesSeq
BasicTypesSeqIn);
// Arrays
        HRESULT Octet_24   ([in,out] tOctet_24    Octet_24In);
        HRESULT Octet_240  ([in,out] tOctet_240   Octet_240In);
        HRESULT Octet_2K4  ([in,out] tOctet_2K4   Octet_2K4In);
        HRESULT Octet_24K  ([in,out] tOctet_24K   Octet_24KIn);
        HRESULT Octet_240K ([in,out] tOctet_240K  Octet_240KIn);

        HRESULT Double_24   ([in,out] tDouble_24    Double_24In);
        HRESULT Double_240  ([in,out] tDouble_240   Double_240In);
        HRESULT Double_2K4  ([in,out] tDouble_2K4   Double_2K4In);
        HRESULT Double_24K  ([in,out] tDouble_24K   Double_24KIn);
        HRESULT Double_240K ([in,out] tDouble_240K  Double_240KIn);

        HRESULT String_24   ([in,out] tString_24    String_24In);
        HRESULT String_240  ([in,out] tString_240   String_240In);
```

```
        HRESULT String_2K4  ([in,out] tString_2K4    String_2K4In);
        HRESULT String_24K  ([in,out] tString_24K    String_24KIn);
        HRESULT String_240K ([in,out] tString_240K   String_240KIn);

        HRESULT BasicTypes_24   ([in,out] tBasicTypes_24    BasicTypes_24In);
        HRESULT BasicTypes_240  ([in,out] tBasicTypes_240   BasicTypes_240In);
        HRESULT BasicTypes_2K4  ([in,out] tBasicTypes_2K4   BasicTypes_2K4In);
        HRESULT BasicTypes_24K  ([in,out] tBasicTypes_24K   BasicTypes_24KIn);
        HRESULT BasicTypes_240K ([in,out] tBasicTypes_240K  BasicTypes_240KIn);
};
```

The IDL-file for the scale-ability benchmark is a lot smaller, as the number of operations on the interface is limited to one. This gives the following IDL-specification

```
// Server.idl : IDL source for Server.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (Server.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";

        [

                uuid(DA091E68-9236-11D1-9641-0060975DB573),

                helpstring("IScaleServer Interface"),
                pointer_default(unique)
        ]
        interface IScaleServer : IUnknown
        {
                [helpstring("method ScaleRequest")] HRESULT ScaleRequest([in] long dw_Value);
        };
[
        uuid(DA091E5B-9236-11D1-9641-0060975DB573),
        version(1.0),
        helpstring("Server 1.0 Type Library")
]
library SERVERLib
{
        importlib("stdole32.tlb");
        importlib("stdole2.tlb");

        [
                uuid(DA091E69-9236-11D1-9641-0060975DB573),
                helpstring("ScaleServer Class")
        ]
        coclass ScaleServer
        {
                [default] interface IScaleServer;
        };
};
```

Appendix B: IIOP data-alignment

Alignment of IIOP-requests on TCP-packets

In order to investigate the size of a chunk of data when sent over the network, a Ethernet-Sniffer has been used. This apparatus can monitor the TCP-packets from all the machines on the network and buffer 3 MB of analysed TCP-packets.

This analysis does a lot more than determining where the packets are coming from and where they are heading. The internals of the packets are also exposed and decomposed for known packet-types. For this small investigation, the data-part of the packets is the most interesting part because there must be the data that is sent from one CORBA component to the other.

For comparison reasons, two data-types where used that have the same target size on the network. Considering the alignment of the data-types included, both should result in 24-bytes of data. The first data-structure is an array of 24-octets, which maps to an array with a size of 24 bytes. The second data-structure is a structure that is composed of different 'basic'-types.

The structure has the following datatypes and members:

type	member-variable	alignment-boundary
double	DoubleVal	8 bytes
float	FloatVal	4 bytes
long	LongVal	4 bytes
short	ShortVal	2 bytes
boolean	BooleanVal	1 bytes
char	CharVal	1 bytes
octet	OctetVal1	1 bytes
octet	OctetVal2	1 bytes
octet	OctetVal3	1 bytes
octet	OctetVal4	1 bytes

From this table it can be seen that the total structure is 24 bytes of data, and because all the data-members are on their alignment-boundaries, they are consecutive (i.e. there are no gaps between the different members to align them)

While both data-sizes are equal, it can be assumed that the size of data-component should be equally-sized. But the Ethernet-Sniffer gave a difference of 6 bytes.
These 6 bytes can be explained as follows:

- the header of the IIOP-request provides an variable space for the name of the interface and the method that should be executed on the server-object. These names are sent over in ASCII-format. Conclusion: The length of the interface & method names have a direct influence on the size of the data in the TCP-packet.
- The alignment of the data-structure members that are transmitted, are not only aligned within the structure, but the alignment of the complete data-structure is aligned on the first natural boundary of the structure with respect to the start of the complete data-buffer in the TCP-packet.

For the array with octets, the first element of the array was positioned at 1 byte after the interface & method name. In front of the structure, there where 5 bytes inserted for aligning the first data-member with the start of the data-buffer of the TCP-packet.

In a figure it looks like the following:

IIOP-request on network (as TCP-packet databuffer)

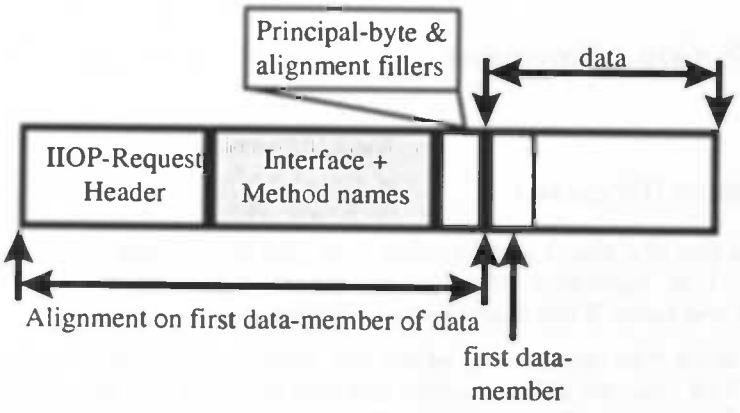


Figure 25:IIOP data-alignment

An example illustrating the loss within data-structures:

```
struct
        {
        octet   val1,
        double  val2,
        octet   val3,
        double  val4,
        ..... etc.
        };
```

This structure will need 7 fill-bytes to close the gap between each octet and consecutive double. For each octet and double that are 9 bytes of useful data, 16 bytes are transmitted. This gives an overhead of:

useful data : (9 / 16) * 100% = 56,25%
overhead : 100 - 56,25 = 43,75%

This example is of course a worse-case situation, but in practice these overheads can easily be ~25%, without noticing. Especially on slow or heavily used networks connections, this can be an issue.

Conclusions:

- By aligning the data-members within structures, the overhead and gap-fillers can be hold minimal, which can give considerable reductions on the network load.
- By considering the alignment inside the data-buffer a small optimisation can be achieved, but this optimisation is very small compared with the size of the data-buffer. The maximum loss can be 7 bytes, whereas 8 bytes is the largest basic-type.

Appendix C: Orbix threading-models

Orbix has supports a multithreading server-objects and clients. For this, Orbix has some additional libraries that can be used safely in the case of a multithreaded program. Although the libraries are thread-safe, the developer himself must ensure thread-safety in his application. To enable the following multithreading options provided with Orbix, the incoming requests on a server, are intercepted before they enter the unmarshalling code of the stub. After the request is intercepted, the request is handed to a specific thread or placed in a queue, depending on the threading model being used.

Thread-per-object

The thread-per-object is a model in which every object has its own thread. Each of these threads accept requests for one object only, and ignores all others. This model can be very useful for real-time processing, where a specific object with its own thread can get a higher priority than other objects and the threads associated with them.

Thread-per-request

The thread-per-request model associates a thread to every incoming request. This means that no call will be blocked by Orbix, but every request will be processed. Although the non-blocking character of this model can appeal, the disadvantages for applications with large numbers of incoming requests, is the large number of threads that must be created and destroyed. In particular for requests in which little processing is required, this creating and destroying of threads can create more overhead than a performance gain.

Thread-per-client

This threading model assigns a thread to every client connected to the server. This means that this thread can execute more than one object, but only on behalf of that single client. This threading model can be very useful for database-transactions, where data-corruption through thread changes can occur. This model has the disadvantage that an idle client, uses resources on the server that cannot be used for other clients or threads.

Pool-of-threads

In this model, a pool of threads is created, that all wait for an incoming request that can be processed. Because the number of threads participating in the pool is fixed, a limit can be set the resources a server needs. Thereby, there is not an excessive creation and destruction of threads compared with the thread-per-request model. The incoming requests are placed in a list or queue, while the threads check this list or queue for new request that can be processed.

Appendix D: Microsoft Transaction Server

Because the Microsoft Transaction Server (MTS) is used during the experiments with DCOM, this appendix describes the architecture and services of the MTS. Because the MTS is an important service for the COM-architecture, not only the functionality that is used during the experiments is described, but a description of its other services is given.

The MTS is a service that is expected to become integrated with the COM-architecture. This, because MTS is built entirely on COM and provides an extension on the normal COM-architecture. The MTS provides services like thread management, efficient object activation policies and support for transactions. Furthermore it provides an efficient re-using mechanism for limited resources like database connections. Before diving into the architecture of MTS and the descriptions of its components, an glossary is given for MTS terminology.

MTS terminology glossary

ActiveX: A set of technologies that enables software components to interact with one another in a networked environment, regardless of the language in which they were created. ActiveX is built upon the Component Object Model.
Administrator: One who use the MTS Explorer to install, configure and manage MTS components and packages.
Business rule: The combination of validation edits, log-on verifications, database lookups, policies and algorithmic transformations that constitute an enterprise's way of doing business. Also known as business logic.
Catalog: The MTS data store that maintains configuration information for components, packages and roles. The catalog is administered by using the MTS Explorer.
Client: An application or process that requests a service from some process or component.
Component: A discrete unit of code built on ActiveX technology that delivers a well-specified set of services through well-defined interfaces. Components provide the server-objects that clients requests at runtime.
Concurrency: The appearance of simultaneously execution of processes or transactions by interleaving the execution of multiple pieces of work.
Context: The state that is implicitly associated with a given MTS object. The context contains information about the object's execution environment, such as the identity of the object's creator and optionally, the transaction encompassing the work of the object. An object's context is similar on concept to the process context that an operating system maintains for an executing program. The MTS runtime environment manages a context for each object.
Declarative security: The security that is configured with the MTS Explorer. Access to packages, components and interfaces is controlled by defining roles. Roles determine which users are allowed to invoke interfaces in a component.
Data source name: The name that applications use to request a connection to an ODBC data source.
Dynamic-link library (DLL): A file that contains one or more functions that are compiled, linked and stored separately form the processes that use them. The operating system maps the DLL's into the address space of the calling process when the process is starting or while it's running.
Identity: A package property that specifies the user accounts that are allowed to access the package. It can be a specific user account or a group of users within a Windows NT domain.

In-doubt transaction: A transaction that has been prepared but hasn't received a decision to commit or abort because the server co-ordinating the transaction is unavailable.
In-process component: A component that runs in a client's process space.
Just-In-Time activation: The ability for an MTS object to be activated only as needed for executing requests form its client. Objects can be deactivated even while clients hold references to them, allowing otherwise idle server resources to be used more productively.
Interface: A group of logically related operations or methods that provides access to a component object.
MTS component: A COM component that executes in the MTS runtime environment. A MTS component must be a DLL, implement a class factory and describe all of the component's interfaces in a type library for standard marshalling.
Microsoft Distributed Transaction Coordinator (DTC): A transaction manager that co-ordinates transactions that span multiple resource managers. Work can be committed as an atomic transaction even if it spans multiple resource managers, potentially on separate computers.
MTS Explorer: A graphical user interface used to configure and manage MTS components within a distributed computer environment.
MTS object: A COM object that executes in the MTS runtime environment and follows the MTS programming and deployment rules.
ODBC resource dispenser: A resource dispenser that manages pools of database connections for MTS components that use the standard ODBC programming interface.
Out-of-process component: A component that runs in a separate process space from its client. MTS enables components implemented as DLL's to be used out-of-process from the client by loading the components into surrogate processes.
Package: A set of components that perform related application functions. All components in a package run together in the same MTS server process. A package is a trust boundary that defines when security credentials are verified. It's also a deployment unit for a set of components. Packages can be created by the MTS Explorer.
Package file: A file that contains information about the components and roles of a package. A package file is created using the package export function of the MTS Explorer. When a package is created, the associated component files (DLL's, type libraries and proxy-stub DLL's) are copied to the same directory where the package file was created.
Pooling: A performance optimisation based on using collections of pre-allocated resources, such as objects or database connections. Pooling results in more efficient resource allocation.
Programmatic security: The procedural logic provided by a component to determine if a client is authorised to perform the requested operation.
Resource dispenser: A service that provides the synchronisation and management of non-durable resources within a process, providing for simple and efficient sharing by MTS objects. For example, the ODBC resource dispenser manages pools of database connections.
Resource dispenser manager: A DLL that co-ordinates work among a collection of resource dispensers.
Resource manager: A system service that manages durable data. Server applications use resource managers to maintain the durable state of the application, such as the record of inventory on hand, pending orders and accounts receivable. The resource managers work in co-operation with the transaction manager to provide the application with a guarantee of atomicity and isolation (using the two-phase commit protocol).
Role: A symbolic name that defines a class of users for a set of components. Each role defines which users are allowed to invoke interfaces on a component.
Server process: A process that hosts MTS components. A MTS component can be loaded into a surrogate server process, either on the client's computer or on another computer. It can also be loaded into a client application process.
Shared property: A variable that is available to all objects in the same server process via the Shared Property Manager. The value of the property can be any type that can be represented by a variant.

State-full object: An object that holds private state accumulated from the execution of one or more client calls.
Stateless object: An object that doesn't hold private state accumulated from the execution of one or more client calls.
Transaction: A unit of work that is done as an atomic operation. This means, that the operation succeeds or fails as a whole.
Transaction context: An object used to allow a client to dynamically include one or more objects in one transaction.
Transaction manager: A system service responsible for co-ordinating the outcome of transactions in order to achieve atomicity. The transaction manager ensures that the resource managers reach a consistent decision on whether the transaction should commit or abort.
Two-phase commit: A protocol ensuring that transactions applying to more that one server are completed on all servers or at none at all. Two-phase commit is co-ordinated by the transaction manager and supported by resource managers.

In the glossary, many parts and services are mentioned. The following sections will put these in the right place. This is done with the help of some pictures, to make the MTS architecture more understandable.

The first pictures shows the architecture of MTS on an very high level, but shows how MTS wraps an ordinary COM-object.

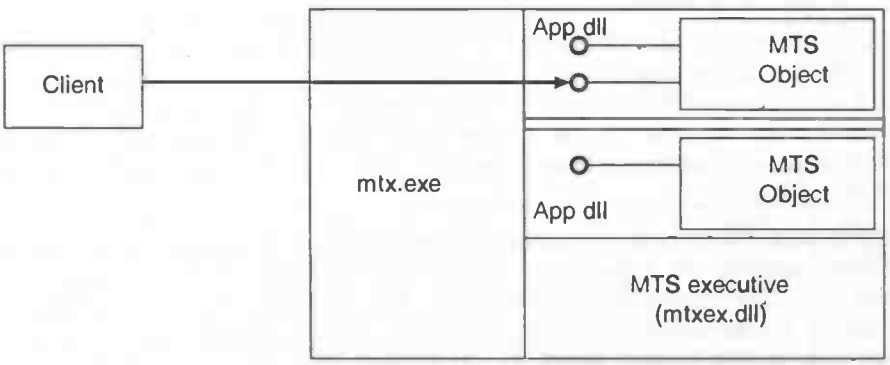


Figure 26:Typical MTS application

From a client's view, the MTS is not noticeable or visible. A client will do its normal creation and binding call, get an interface pointer and start invoking requests on the COM-object. Because the COM-object is inserted in a MTS package (and is called an MTS-object), the MTS will catch any request a client invokes. The MTS will launch the COM-object and creates an context-object for it, after which the client's request is passed to the COM-object. This situation in visualised in the following picture.

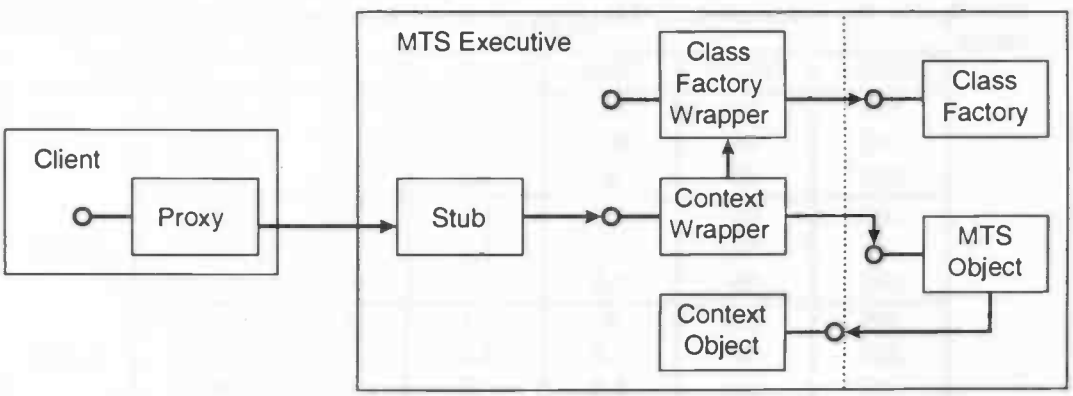


Figure 27:How MTS wraps objects

In this situation, the MTS object has an reference to its Context Object, which provides a set of functions. Among these functions, are the SetAbort and SetComplete functions.

When an object calls SetAbort, all the transactions the object made since the last SetComplete (when it was doing transactions at all), are all rolled back. The SetComplete enforces the committing of the transactions that are made. Besides the committing of the transactions, the SetComplete informs the MTS Executive that the object has no relevant state. This can be due to the fact that the object never has client-specific state at all, or that all the relevant was committed in a resource dispenser. Depending on the fact, that an object told MTS that it can 'pooled', the MTS Executive keeps the object alive, or deactivates the object and release its resources. The situation when a deactivation takes place, is shown in the following picture:

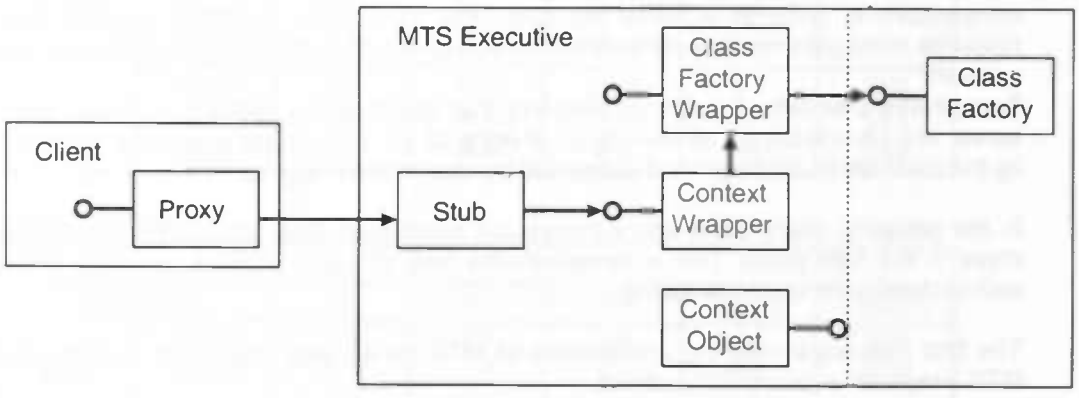


Figure 28:Deactivated MTS Object

When the client invokes a new request, the MTS Executive uses the class factory to create a new object. For the client, this is an invisible process, and because the old object had no state it is indistinguishable from the newly created one. By calling SetComplete as often as possible, applications can be scaled much better. The process of an object calling SetComplete, its deactivation and creating new objects when needed is called, Just-In-Time Activation.

For MTS objects that use sparse resources, like connections to databases, an equivalent resource-scheme applies. An object that invokes transactions on a database, likes to keep its connection open during its entire life-time. This is due to the fact, that creating a new connection to the database costs a considerable amount of time. On the other side, database connections are sparse resources, from which actually small numbers can be maintained. When building applications with large numbers of clients and therefore MTS-objects, this can become a problem due to the number of available database connections. The MTS uses an advanced technique, by defining resource-dispensers. A resource-dispenser provides an efficient way for freeing and acquiring shared, non-persistent resources. The ODBC dispenser for instance, maintains a pool of database connections. These database connections can than be used by the MTS objects, which are encouraged to create and destroy connections to the database (which is actually the pool managed by resource dispenser) as much as possible. Using this approach, the MTS objects use the resource as efficient as possible, while enabling applications to scale better.

Appendix E: Results marshalling benchmarks

For both of the middleware products, four matrices are given.

- data-type : array
  interface-type : IN
- data-type : sequence
  interface-type : IN
- data-type : array
  interface-type : IN,OUT
- data-type : sequence
  interface-type : IN,OUT

These matrices contain the mean, 5% lower-bound and 95% upper-bound limits of the response-times measured. This means that the intervals between the lower- and upper-bound represent the interval of 90% of all response times. This interval must be considered as an empirical 90% confidence interval.

Result matrices CORBA/Orbix

ARRAYS

IN		data-size				
data-type		24	240	2400	24000	240000
octet	upper	3.1	4.9	8.8	29	262
	mean	3	3.9	5.8	26	243
	lower	2.9	3.1	5	25	236
double	upper	3.9	4.1	6.3	31	253
	mean	3.2	3.5	5.3	26	239
	lower	2.9	3.1	5	25	233
struct	upper	3.2	3.3	5.8	40	304
	mean	3.1	3.3	5.7	32	291
	lower	2.9	3.1	5.5	30	283

SEQUENCES

IN		data-size				
data-type		24	240	2400	24000	240000
octet	upper	4.2	5.9	16	53	357
	mean	3.3	3.8	7.8	40	342
	lower	2.9	3.2	6	35	336
double	upper	3.1	3.4	5.4	29	262
	mean	3	3.2	5.3	27	251
	lower	2.9	3.1	5.1	26	245
struct	upper	3.2	4.3	6	32	295
	mean	3.1	3.4	5.7	30	288
	lower	2.9	3.1	5.5	30	282

ARRAYS

IN-OUT		data-size				
data-type		24	240	2400	24000	240000
octet	upper	3.4	3.6	8	51	526
	mean	3.1	3.4	7.4	48	489
	lower	2.9	3.2	7.1	46	463
double	upper	3.1	3.6	7.5	52	487
	mean	3.1	3.5	7.4	48	467
	lower	2.9	3.3	7.1	46	456
struct	upper	25	4.5	14.4	71	713
	mean	10	4.5	10	61	591
	lower	3	3.4	8.2	56	552

SEQUENCES

IN-OUT		data-size				
data-type		24	240	2400	24000	240000
octet	upper	3.3	3.7	9.8	84	729
	mean	3.1	3.6	9.6	72	683
	lower	2.9	3.5	9.2	67	663
double	upper	3.3	3.6	8.6	55	636
	mean	3.1	3.5	7.8	51	522
	lower	2.9	3.3	7.4	49	485
struct	upper	6.1	7.1	28	181	751
	mean	4	4.5	13	82	616
	lower	3.1	3.6	8.3	56	558

Result matrices DCOM

ARRAYS

IN		data-size				
data-type		24	240	2400	24000	240000
octet	upper	0.95	11.7	6.7	27	256
	mean	0.89	2.6	4.1	24	245
	lower	0.86	1.1	3.1	23	239
double	upper	1.3	1.1	4	28	266
	mean	1.1	1.1	3.4	24	248
	lower	0.85	1.1	3.2	24	239
struct	upper	3	1.2	4.2	33	251
	mean	1.4	1.2	4	32	243
	lower	0.85	1.1	3.9	31	239

SEQUENCES

IN		data-size				
data-type		24	240	2400	24000	240000
octet	upper	1	1.2	3.5	28	311
	mean	0.97	1.1	3.2	25	254
	lower	0.86	1.1	3.1	24	239
double	upper	1	1.1	3.2	31	254
	mean	0.95	1.1	3.2	25	245
	lower	0.85	1.1	3.2	24	239
struct	upper	0.94	1.1	3.2	25	248
	mean	0.89	1.1	3.2	24	242
	lower	0.85	1	3.1	24	239

ARRAYS

IN-OUT		data-size				
data-type		24	240	2400	24000	240000
octet	upper	1.2	1.3	5.9	55	580
	mean	0.94	1.3	5.6	51	530
	lower	0.87	1.3	5.4	49	511
double	upper	0.93	1.3	5.6	53	533
	mean	0.92	1.3	5.5	50	520
	lower	0.87	1.3	5.4	49	511
struct	upper	1.2	1.4	8	69	594
	mean	1	1.4	6.5	59	531
	lower	0.88	1.3	6.2	58	511

SEQUENCES

IN-OUT		data-size				
data-type		24	240	2400	24000	240000
octet	upper	1	1.3	5.9	52	558
	mean	0.95	1.3	5.6	50	529
	lower	0.9	1.3	5.5	49	511
double	upper	1	1.3	5.7	53	557
	mean	1	1.3	5.6	50	533
	lower	0.9	1.3	5.5	49	512
struct	upper	1.5	1.5	9.2	61	584
	mean	1.1	1.4	6.4	53	529
	lower	0.9	1.3	5.5	49	510

Appendix F: Results scale-ability benchmarks

This appendix gives two matrices in which the results are presented which are obtained during the execution of the benchmarks belonging to experiments F,H and L.

The matrices contain the mean, 5% lower-bound and 95% upper-bound limits of the response-times measured. This means that the intervals between the lower- and upper-bound represent the interval of 90% of all response times. This interval must be considered as an empirical 90% confidence interval.

The results for CORBA/Orbix

CORBA		nr. of lightweight processes per client capsule								
nr. of PC's		1	2	3	4	5	10	15	20	25
1	upper	4.2	6.2	21.6	12.3	12.4	34	53	73	94
	mean	3.9	4.6	9.7	8.2	8.2	24	43	61	80
	lower	3.8	3.8	4.1	4.1	4.2	11	32	50	66
5	upper	16	40	61	83	108	219	316	407	506
	mean	8	27	46	65	86	182	277	370	469
	lower	4	15	33	50	68	156	251	342	439
10	upper	38	80	140	199	249	444	651	853	996
	mean	28	66	105	144	183	372	571	762	924
	lower	18	53	72	110	149	331	527	717	877
15	upper	56	121	226	263	299	629	885	1170	1444
	mean	45	101	155	211	269	542	814	1099	1375
	lower	36	63	115	188	246	502	773	1055	1327
20	upper	76	197	231	307	385	773	1157	1546	1917
	mean	63	131	211	284	361	737	1116	1493	1860
	lower	51	75	194	264	338	707	1078	1451	1812
25	upper	97	203	294	385	487	965	1448	1926	2428
	mean	82	176	271	361	459	929	1404	1869	2365
	lower	68	157	251	339	434	895	1366	1818	2303
30	upper	113	233	351	466	586	1167	1750	2313	2914
	mean	100	214	327	438	556	1126	1700	2256	2842
	lower	87	198	307	415	527	1087	1657	2206	2781
40	upper	152	313	465	628	780	1571	2354	3105	3913
	mean	115	292	440	599	747	1522	2297	3042	3836
	lower	48	274	417	571	716	1476	2243	2979	3769

Figure 29:Orbix, result matrix experiment L

The results for DCOM.

DCOM		nr. of lightweight processes per client capsule								
nr. of PC's		1	2	3	4	5	10	15	20	25
1	upper	1.7	1.9	2.1	2.3	2.4	3.3	4.5	5.8	7
	mean	1.6	1.7	1.7	1.8	2	2.4	3	3.6	4.3
	lower	1.5	1.5	1.5	1.6	1.6	1.8	1.9	2	2.3
5	upper	7	37	3.2	50	38	60	118	128	157
	mean	2.5	8.7	2.7	9.9	7.9	19	36	52	70
	lower	1.5	1.5	1.6	1.6	1.7	4.4	12	21	27
10	upper	2.6	4.1	8.2	16	23	63	111	131	151
	mean	1.8	2.3	4.2	8.8	13	40	72	90	113
	lower	1.5	1.5	1.7	3	2.2	21	38	54	81
15	upper	3.2	8.4	23	34	39	97	152	178	213
	mean	2	3.6	11.7	19	23	66	105	143	180
	lower	1.5	1.7	3.7	8.1	5.7	34	51	96	152
20	upper	4.5	16	31	44	59	132	182	234	303
	mean	2.3	8.7	19	29	39	94	142	192	240
	lower	1.5	2.6	8.1	13	13	47	92	154	170
25	upper	3.8	22	41	59	83	150	208	292	363
	mean	2.1	12.1	25	38	54	113	168	226	297
	lower	1.5	2.8	11	20	25	73	113	120	187
30	upper	9.9	30	52	73	100	185	263	343	457
	mean	4.1	18	34	49	64	139	200	280	374
	lower	1.6	7	14	23	36	95	125	222	291
40	upper	17.1	46	76	106	126	220	390	558	679
	mean	8.1	28	48	70	90	186	275	401	585
	lower	1.7	10	22	35	50	163	130	159	517
50	upper	24	67	94	120	157	300	588	745	
	mean	13	39	62	86	117	236	374	564	
	lower	3.5	15	29	39	56	180	127	383	

Figure 30:DCOM, result matrix experiment L

Appendix G: SPSS output on multiple regressions

The following output was generated for the multiple regression discussed in the result evaluation of experiment L. First the output for both middleware products is given for linear regression, where-after the output for the non-linear multiple regression is given.

Regression

Descriptive Statistics			
	Mean	Std. Deviation	N
RESPONSE	641,1055556	821,9099531	72
PC	18,25	12,39235339	72
PC_THREA	172,3611111	215,8109048	72
THREADS	9,444444444	8,237476861	72

Correlations					
		RESPONSE	PC	PC_THREA	THREADS
Pearson Correlati	RESPONSE	1	0,5414889	0,999889611	0,69373865
	PC	0,5414889	1	0,542321498	3,32402E-18
	PC_THREA	0,999889611	0,542321498	1	0,696600354
	THREADS	0,69373865	3,32402E-18	0,696600354	1
Sig. (1-tailed)	RESPONSE	.	4,52415E-07	2,73983E-18	1,90982E-12
	PC	4,52415E-07	.	4,32028E-07	0,5
	PC_THREA	2,73983E-18	4,32028E-07	.	1,31118E-12
	THREADS	1,90982E-12	0,5	1,31118E-12	.
N	RESPONSE	72	72	72	72
	PC	72	72	72	72
	PC_THREA	72	72	72	72
	THREADS	72	72	72	72

Model Summary						
	Variables		R	R Square	Adjusted R Squar	Std. Error of the Estimate
Model	Entered	Removed				
	THREADS, PC, P	.	0,999906399	0,999812807	0,999804548	11,49063201
a	Dependent Variable: RESPONSE					
b	Method: Enter					
c	Independent Variables: (Constant), THREADS, PC, PC_THREA					
d	All requested variables entered.					

Coefficients								
		Unstandardized Coefficients		Standardized Coefficients		Sig.	95% Confidence Interval for B	
Model		B	Std. Error	Beta			Lower Bound	Upper Bound
1	(Constant)	-4,676649285	[illegible]		-1,264058535	0,21052495	-12,05930457	2,706006001
	PC	-0,435819248	0,168081814	-0,006571068	-2,592899478	0,011642089	-0,771221288	-0,100417207
	PC_THREA	3,848771384	0,013452525	1,010581307	286,1002887	6,30019E-18	3,821927287	3,87561548
	THREADS	-1,020981812	[illegible]	-0,010232647	-3,448000307	0,000973428	-1,611856904	-0,430106719
a	Dependent Variable: RESPONSE							

Figure 31:Orbix, linear regression output

Regression

Descriptive Statistics			
	Mean	Std. Deviation	N
RESPONSE	89,685	124,7917192	80
PC	21,425	15,16222404	80
PC_THREA	192,625	236,7348768	80
THREADS	9,25	8,093598033	80

Correlations					
		RESPONSE	PC	PC_THREA	THREADS
Pearson Correlati	RESPONSE	1	0,500777841	0,99425301	0,664602886
	PC	0,500777841	1	0,526513782	-0,045850084
	PC_THREA	0,99425301	0,526513782	1	0,671134099
	THREADS	0,664602886	-0,045850084	0,671134099	1
Sig. (1-tailed)	RESPONSE	.	1,11919E-06	1,62995E-18	3,8012E-12
	PC	1,11919E-06	.	2,63418E-07	0,343160108
	PC_THREA	1,62995E-18	2,63418E-07	.	1,71364E-12
	THREADS	3,8012E-12	0,343160108	1,71364E-12	.
N	RESPONSE	80	80	80	80
	PC	80	80	80	80
	PC_THREA	80	80	80	80
	THREADS	80	80	80	80

Model Summary						
	Variables		R	R Square	Adjusted R Squar	Std. Error of the Estimate
Model	Entered	Removed				
	THREADS, PC, P	.	0,994965475	0,989956297	0,989559835	12,7508594
a	Dependent Variable: RESPONSE					
b	Method: Enter					
c	Independent Variables: (Constant), THREADS, PC, PC_THREA					
d	All requested variables entered.					

Coefficients								
		Unstandardized Coefficients		Standardized Coef	t	Sig.	95% Confidence Interval for B	
Model		B	Std. Error	Beta			Lower Bound	Upper Bound
1	(Constant)	-0,828259772	3,780469634		-0,219089122	0,827167943	-8,357717651	6,701198107
	PC	-0,468813791	0,143820353	-0,056960989	-3,259717987	0,001669801	-0,755256851	-0,182370731
	PC_THREA	0,556276323	0,012412206	1,055278409	44,81687883	3,69184E-18	0,531555273	0,580997374
	THREADS	-0,712987196	0,308980158	-0,046242105	-2,307550102	0,023747554	-1,328374518	-0,097599873
a	Dependent Variable: RESPONSE							

Figure 32:DCOM, linear regression output

Output of SPSS for non-linear multiple regression:

Non-linear Regression

```
All the derivatives will be calculated numerically.

The following new variables are being created:

Name          Label

PRED_         Predicted Values
RESID         Residuals
```

Iteration	Residual SS	B0	B1	B2
1	77556229,94	,000000000	,000000000	,000000000
1.1	74943424,72	,267826011	,518763349	,028666668
2	74943424,72	,267826011	,518763349	,028666668
2.1	70047981,94	,781953424	1,51645897	,084081623
3	70047981,94	,781953424	1,51645897	,084081623
3.1	61433500,21	1,73029065	3,36385312	,187780311
4	61433500,21	1,73029065	3,36385312	,187780311
4.1	46093686,46	3,59028633	7,02202317	,398620504
5	46093686,46	3,59028633	7,02202317	,398620504
5.1	22917879,30	7,05336517	14,0626910	,843790030
6	22917879,30	7,05336517	14,0626910	,843790030
6.1	3915784,957	8,42679198	19,3917850	2,06749998
7	3915784,957	8,42679198	19,3917850	2,06749998
7.1	9189,325166	-,61197799	-1,3039109	3,85942869
8	9189,325166	-,61197799	-1,3039109	3,85942869
8.1	9189,325166	-,61197789	-1,3039106	3,85942866

```
Run stopped after 16 model evaluations and 8 derivative evaluations.
Iterations have been stopped because the relative reduction between
successive
residual sums of squares is at most SSCON = 1,000E-08
```

Nonlinear Regression Summary Statistics Dependent Variable RESPONSE

Source	DF	Sum of Squares	Mean Square
Regression	3	77547040,6148	25849013,5383
Residual	69	9189,32517	133,17863
Uncorrected Total	72	77556229,9400	
(Corrected Total)	71	47963053,9378	

R squared = 1 - Residual SS / Corrected SS = ,99981

Parameter	Estimate	Asymptotic Std. Error	Asymptotic 95 % Confidence Interval Lower	Upper
B0	-,611977887	,094376502	-,800253857	-,423701917
B1	-1,303910577	,194699267	-1,692324987	-,915496167
B2	3,859428661	,010528092	3,838425692	3,880431629

Asymptotic Correlation Matrix of the Parameter Estimates

	B0	B1	B2
B0	1,0000	,0000	-,5423
B1	,0000	1,0000	-,6966
B2	-,5423	-,6966	1,0000

Figure 33:Orbix, non-linear multiple regression output

Non-linear Regression

All the derivatives will be calculated numerically.

The following new variables are being created:

Name	Label
PRED_	Predicted Values
RESID	Residuals

Iteration	Residual SS	B0	B1	B2
1	1873736,820	,000000000	,000000000	,000000000
1.1	1495715,819	,206588315	,025585500	,497160247
2	1495715,819	,206588315	,025585500	,497160247
2.1	886784,6330	,598484638	,078759763	1,46636926
3	886784,6330	,598484638	,078759763	1,46636926
3.1	222900,4103	1,05583599	,216987209	2,92206566
4	222900,4103	1,05583599	,216987209	2,92206566
4.1	20096,77048	-,14490697	,490150612	,270105377
5	20096,77048	-,14490697	,490150612	,270105377
5.1	12364,21963	-,49458891	,557907528	-,76406400
6	12364,21963	-,49458891	,557907528	-,76406400
6.1	12364,21963	-,49458891	,557907540	-,76406433

Run stopped after 12 model evaluations and 6 derivative evaluations.
Iterations have been stopped because the relative reduction between successive
residual sums of squares is at most SSCON = 1,000E-08

Nonlinear Regression Summary Statistics Dependent Variable RESPONSE

Source	DF	Sum of Squares	Mean Square
Regression	3	1861372,60037	620457,53346
Residual	77	12364,21963	160,57428
Uncorrected Total	80	1873736,82000	
(Corrected Total)	79	1230264,88200	

R squared = 1 - Residual SS / Corrected SS = ,98995

Parameter	Estimate	Asymptotic Std. Error	Asymptotic 95 % Confidence Interval Lower	Upper
B0	-,494588909	,082212754	-,658295418	-,330882401
B1	,557907528	,009869610	,538254624	,577560431
B2	-,764064003	,201517397	-1,165336405	-,362791601

Asymptotic Correlation Matrix of the Parameter Estimates

	B0	B1	B2
B0	1,0000	-,5689	,0429
B1	-,5689	1,0000	-,6972
B2	,0429	-,6972	1,0000

Figure 34:DCOM, non-linear multiple regression output

Appendix H: Program outline in scale-ability benchmarks

Figure 35 gives an a brief outline for the communication-protocol between the server, the client and the synchronisation server in the experiments with a large number of client-nodes. The client-node in this figure has multiple lightweight processes, but the main process handles the synchronisation on behalf of the lightweight processes that invoke the requests on the server.

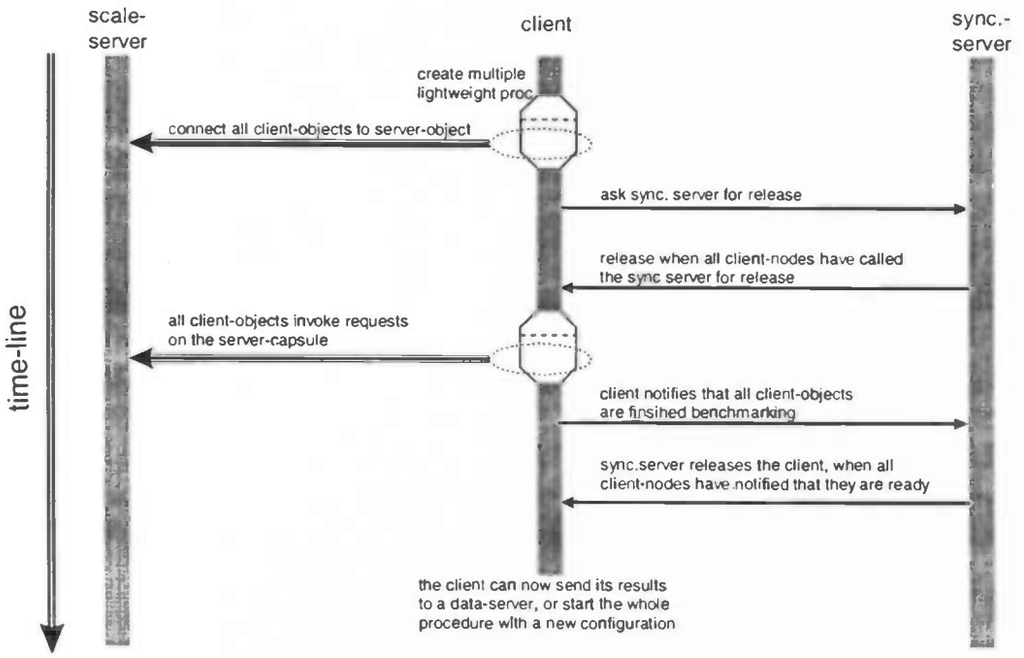


Figure 35:Program outline in scale-ablity benchmarks