Master Thesis
Software Engineering
Thesis no: MSE-2003:22
06 2003

Performance Analysis of Distributed Object Middleware Technologies

Richard Bladh & Per Arneng

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 2*20 weeks of full time studies.

Contact Information:
Author(s):
Richard Bladh
Per Arneng
E-mail: Richard Bladh <pt99rbl@student.bth.se> , Per Arneng <pt99par@student.bth.se>

University advisor(s):
Håkan Grahn
Department of Software Engineering and Computer Science

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Internet : http://www.bth.se/ipd
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Each day new computers around the world connects to the Internet or some network. The increasing number of people and computers on the Internet has lead to a demand for more services in different domains that can be accessed from many locations in the network.

When the computers communicate they use different kinds of protocols to be able to deliver a service. One of these protocol families are remote procedure calls between computers. Remote procedure calls has been around for quite some time but it is with the Internet that its usage has increased a lot and especially in its object oriented form which comes from the fact that object oriented programming has become a popular choice amongst programmers. When a programmer has to choose a distributed object middleware there is a lot to take into consideration and one of those things is performance.

This master thesis aims to give a performance comparison between different distributed object middleware technologies and give an overview of the performance difference between them and make it easier for a programmer to choose one of the technologies when performance is an important factor. In this thesis we have evaluated the performance of CORBA, DCOM, RMI, RMI-IIOP, Remoting-TCP and Remoting-HTTP.

The results we have seen from this evaluation is that DCOM and RMI are the distributed object middleware technologies with the best overall performance in terms of throughput and round trip time. Remoting-TCP generally generates the least amount of network traffic, while Remoting-HTTP generates the most amount of network traffic due to it's SOAP-formated protocol.

Keywords: Distributed Programming, Performance Analysis, Object Orientation, Distributed Object Middleware

ACKNOWLEDGEMENTS

The authors of this thesis would like to thank the following:

- Håkan Grahn for being a really good supervisor.
- Blekinge Institute of Technology.

TABLE OF CONTENTS

1 Introduction

1.1 Motivation for writing this thesis

With the increasing numbers of computers that get connected to local area networks and the Internet every day the demand increases for applications that can communicate with other applications or servers. One of the communication types that applications use to communicate is distributed object middleware which is a way to make method calls on a remote computer via a local proxy object and making it look like a local method call on an ordinary local object.

Many software projects has performance as an important quality attribute and since distributed object middleware affects the overall performance of an application it is then important to know how the different distributed object middleware relate to each other in terms of performance.

So the main reason why we have decided to look at performance in different distributed object middleware is just because of the increasing use of network communication between computers and the fact that distributed object middleware is one technology which is used for computer communication today and probably will be used more in the future.

1.2 Problem definition

When creating an application which uses distributed object middleware to communicate with other applications you have to make a choice on which distributed object middleware to use based on different quality requirements on the application. Making this choice can either be made by experience or by looking at some evaluations made by someone else. With this thesis we aim to make this problem easier to solve by giving some decision guidance about which distributed object middleware technology to choose when performance is the main quality requirement.

With this thesis we would like to gain knowledge about the performance of distributed object middleware technologies in order to answer the following questions:

- How much network traffic do a specified distributed object middleware technology generate when calling a specified method and how do the different technologies compare to each other in terms of network traffic per method call?
- How do the distributed object middleware technologies compare against each other in terms of scalability and throughput?

1.3 Scope and Limitations

All the tests we perform are done on the virtual platforms .NET and SUN's Java platform so the performance results we have gathered here should only be considered to be valid on those platforms.

In some of the distributed object middleware technologies we have tested there are some optimizations that can be done and then making it perform better. Due to limitations in time we have chosen to test the technologies with their default configurations, also if we made some configuration that would optimize one technology we would have to do it on the others as well.

This would in the end result in an unfair test since there are essentially hundreds of parameters that could be set per technology to make the run more efficiently and perform better.

The test environment is between two computers since we have been unable to find an environment where we could test it with multiple computers. To simulate a multicomputer environment we use multiple threads that acts like clients towards the server.

The results would probably have differed somewhat if we used multiple clients instead of threads, but due to time constraints and the lack of a suitable test environment we chose to conduct the tests using threads instead. Also, most of the tests that we looked on were using threads as well.

If we had chosen clients instead we would have needed to consider facts like the need for equally powerful computers and equally performance over the network wire (things we cannot control).

There are also a great number of distributed object middleware technologies available today. We have only chosen the technologies that comes with the two virtual platforms by default. These technologies are:

- CORBA
- RMI
- RMI-IIOP
- DCOM using TCP
- .NET Remoting using TCP
- .NET Remoting using HTTP

1.4 Methodology

There are three things we have decided to test in our master thesis. The first one we have chosen is to measure the round trip time it takes for a method call in an environment where we have multiple similar calls at the same time. The round trip time is measured on each thread and the results we uses is the average of this time over all the threads in a test.

We are also going to look at the throughput in method calls per milliseconds which shows how good or bad a distributed object middleware server performs and scales.

The other one is to measure the total amount of data that is transferred between the client and the server when making a single method call with different parameters. We are also going to look at the relationship between the actual data size of the method parameters compared to the total size of the data that is transferred during a method call.

1.5 Main Contribution

Our main contribution is that we can deliver a performance comparison between six different distributed object middleware in terms of throughput, scalability and network load. This thesis can act as a guide for those who wish to chose a distributed object middleware for use with their application when performance is the highest prioritized quality attribute.

The thesis can also be interesting for those who use applications on networks where you have to pay for the amount of data you transfer or where the network bandwidth is very small. In those cases the total network load per method call can be very important.

1.6 Thesis Outline

Part One

Contains information and background about the thesis itself along with motivation and methodology.

Part Two

This part contains background of the distributed object middleware we have decided to test and motivation on why we have selected the specific technologies and then a summary of some of the similarities and differences between them.

Part Three

Contains information about the test framework and the setup. There is also information about the test environment and how the tests where made.

Part Four

This part contains the actual results from the tests we have done. This part is divided in to a part which shows the results from the data size analyze and a part which shows the throughput and scalability results.

Part Five

This part contains a discussion where we discuss things and thoughts we have had during this thesis and the tests we have done.

Part Six

This part contains information about related work.

Part Seven

This part contains information about future work related to the contents of this thesis.

Part Eight

This part contains the conclusion.

1.7 Definitions

In-process

In-process server objects are implemented in a dynamic-link library (DLL) and are run in the process space of the controller or parent application (in our case the COM-runtime environment). Because they are contained in a DLL, they cannot be run as stand-alone objects.

Access to in-process objects is much faster than to out-of-process server objects since remote procedure calls are within the same process boundary.

Out-of-process

Out-of-process server objects are implemented in an executable file and are run in a separate process space.

Overhead

The total size of network data transferred to and from the client minus the actual data. This includes meta-data about the invoked method as well as other information related to the call thats being made.

Actual data

The data that is specific for the data type sent as a parameter or return value, i.e. for a double that is 64 bits or 8 byte, 0 bytes for the empty method call.

Scalability

How well a certain distributed object middleware performs when changing the number of clients connected to the server.

Middleware

Middleware is according to Rock-Evans; “Middleware is 'off the shelf connectivity software, which supports distributed processing' at runtime and which is 'used by developers to build distributed software” [Rock-Evans98].

Stubs

Piece of code that acts like a proxy to the remote object. Method calls on this object forwards the parameters to the remote method and at the same time making it look like an ordinary local method call.

Marshalling

This is the process of passing parameters from one context to another. When using distributed object middleware this means serializing the parameters from and to a transmittable format.

Singleton factories

Follows the abstract factory design pattern and is essentially an singleton instance of a class that contains a list of implementations that can only be instantiated through parameter calls to the singleton instance. It is primarily used to decouple functionality and parts in a software application.

2 Background

2.1 Remote Procedure Calls

2.1.1 Overview

Remote Procedure Calls (RPC) is a system that enables a programmer to create applications that can call procedures that exists on other machines. RPC make use of proxy methods which has the same method signature as the remote method but contains code for transferring data between the client and the server while making it look like a normal local method call. The RPC facility handles all the communications to and from the remote machine. It takes the parameters and bundles them together and sends them to the remote machine where they are unpacked and send to the requested method. The same happens with the return value from the requested method. It is packed down and sent to the client and then unpacked and returned as if it was returned from a local procedure.

RPC really simplifies the task of creating means of communications between networked entities. When working with RPC the programmer decouples him/her self from handling low level network programming. [Downing98]

2.1.2 Passing Parameters

Most programming languages allows the programmers to pass the method parameters by value or pointers to locations where the value is stored.

When dealing with remote procedures it is not that simple. Passing parameters by value is simple because it is just a matter of marshaling the parameter down to a network transmittable representation. When passing a parameter by reference is harder because you will need some kind of system wide pointer for each parameter. The overhead created with these kinds of references might not be worth the effort. [Stallings98]

2.1.3 Parameter Representation

When communicating between computers it is important that the data that is sent arrives in the same format as when it was sent. This is easy in a homogeneous environment. There are problems when machines and applications are different on the endpoints of the communication. The things that often differ are the representation of data types and text strings.

To solve this problem a standardized format for common data such as integers, strings and floating point numbers has to be created. Then the data formats on the native machine and application can be transformed from and to this standardized format. [Stallings98]

2.1.4 Object Oriented Approach

In today's modern operating systems there are more and more focus on object oriented technologies. The object orientation is now the modern way to go when developing applications. Even databases are gradually moving towards an object oriented approach. Client/server designers have also begun to embrace this approach. In a client/server context this means passing objects between the client and the server. This can be done by building on

top of an already existing RPC mechanisms or using object oriented mechanisms in the operating system. [Stallings98]

Just passing objects as messages back an fourth are not always the best solution for our problems. We would need a mechanism similar to RPC but with passing objects as parameters and return values. Introducing RPC with object orientation adds a lot of extra complexity to the mechanism but it also adds better integration with object oriented programming languages and operating systems. [Downing98]

2.2 .NET Remoting

2.2.1 Motivation

.NET Remoting is the latest major distributed infrastructure which challenges the predecessors of distributed application technologies. Among other things .NET Remoting is powering the Web Services framework from Microsoft which is widely used and hyped these days. Although, Web Services has a simplified programming model and are intended for a wide target audience.

The main reason why we have chosen this technology is because it's a part of the .NET framework which is a relatively new technology. Another reason is the fact that it's highly extensible because it supports custom communication protocols and protocol formats. This fact makes .NET Remoting a worthy contestant in the choice of a distributed infrastructure today and in the future.

2.2.2 Overview

.NET Remoting is the latest distributed infrastructure from Microsoft that provides a rich set of classes that allow developers to ignore most of the complexities of deploying and managing remote objects.

Distributed application technologies such as DCOM, CORBA, RMI and RMI-IIOP have evolved over many years to keep up with the constantly increasing requirements of the enterprise. This has made them large and somewhat complex.

Every so often, it's better to start over from the beginning which is what the designers of .NET Remoting did. The designers of .NET Remoting had the advantage of taking into account the recent technology requirements that the designers of DCOM, CORBA, RMI and RMI-IIOP initially didn't have. This leaves .NET Remoting slim, uncluttered and streamlined opposed to the other major distributed frameworks.

As with most of the remoting frameworks, .NET Remoting abstracts the call of methods on remote objects to be nearly identical to calling a local method.

One big advantage with the .NET Remoting framework is that it supports multiple communication protocols and protocol formats, thus enabling the application to communicate over platform boundaries. .NET Remoting also supports custom created communication protocols and protocol formats, which in practice allows .NET Remoting to be adaptable to the network environment in which it is being used. But one should keep in mind that the pure version of .NET Remoting requires the clients to be built using .NET, or another framework that supports .NET Remoting, which means a homogeneous environment. If there is a need for platform independence then Web Services is .NET Remoting:s answer. To work this way Web Services use Simple Object Access Protocol, or SOAP to communicate. SOAP is an open and Internet-based standard protocol based on the Extensible Markup Language, or XML-standard.

2.2.3 How .NET Remoting works

Short Overview

When a client creates an instance of a known remote object, it receives a proxy to the class instance or object on the server. All methods called on the proxy will automatically be forwarded to the remote class and any results will be returned to the client. From a client's point of view, this process is no different from making a local call.

In practice that means;

1. A remote object is registered on the server or producer-machine in a proxy-interface and a port and name of the service is specified.
2. The object is marshaled so that it contains all information about where to find the specific object instance on the network.
3. The client opens up a channel of a specified type (standard choices are TCP and HTTP) over which the communication will be done.
4. The client or consumer of a "service" retrieves the remote object through a call to the remote proxy.
5. The client activates the object using the meta-data (runtime type information) through the Activator-functionality in .NET Remoting.
6. The client accesses the functionality on the object as if it were local.

Distributed Objects in .NET Remoting

"Any object outside the application domain of the caller should be considered remote, even if the objects are executing on the same machine." [Obermeyer03]

Inside the application domain, all objects are passed by reference while primitive data types are passed by value. And since local object references are only valid inside the application domain where they were created, they cannot be passed to or returned from remote method calls in that form. They need to be serialized.

Note that; "Local objects that cannot be serialized cannot be passed to a different application domain and are therefore non-remotable." [McLean03]

2.2.4 Remote Object Proxies

When a client activates a remote object in obtains a proxy to it. "The proxy object acts as a representative of the remote object and ensures that all calls made on the proxy are forwarded to the correct remote object instance." [McLean03].

Every operation on this proxy is appropriately indirected to enable the remoting infrastructure to intercept and forward the calls. The indirection do have some impact on performance, but the .NET JIT compiler and execution engine are optimized to prevent unnecessary performance penalties when the proxy and remote object reside in the same application domain, just as for DCOM [Rammer02].

Actually, there are 2 proxies created on the client-side, a TransparentProxy and a RealProxy.

The TransparentProxy is created in runtime and is an exact mapping of the remote objects interface. This is the proxy that the client accesses directly. When the client interacts with the TransparentProxy an message object is created which is forwarded to the RealProxy as can be seen in figure 1.

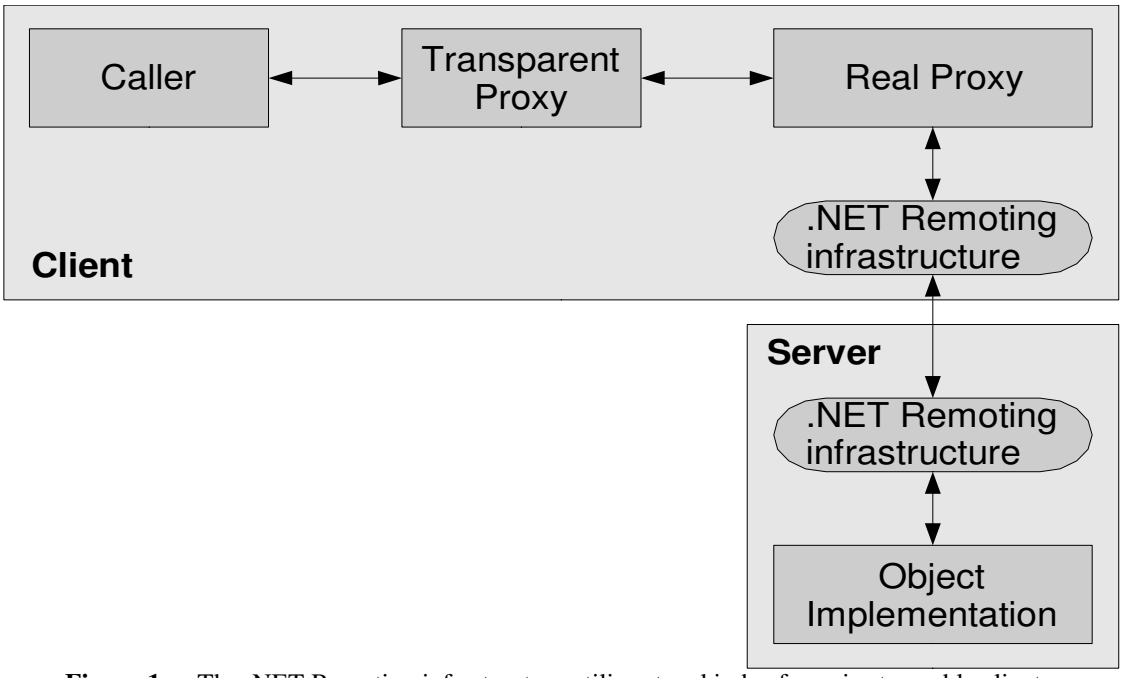

Figure 1. The .NET Remoting infrastructure utilizes two kinds of proxies to enable clients to interact with remote objects.

The RealProxy-instance is the one that routes the actual call through the NET Remoting infrastructure to the remote machine that hosts the object. [MS_RemotingSpec03]

2.2.5 Easy Deployment of Distributed Objects

All .NET objects can be installed on new machines with an easy XCopy deployment. [Tallamraju03] What XCopy deployment really means is that it's as simple as just copying the files to another location and run it directly without further ado. Therefore no registration whatsoever is needed which makes it totally independent of extra software except for the .NET Framework.

Windows 3.11 and Windows 95 utilized the concept of ini-files for each application. Ini-files contained information about how to setup the application properly. Later on, Microsoft decided to introduce the Windows Registry which would centralize the meta-data that were needed for the applications, thus making ini-files obsolete.

In .NET the concept of ini-files are back in the form of XML-files that describes the meta-data, such as remoting information, security settings, object deployment and server setup.

2.2.6 Remoting-TCP

Remoting-TCP uses TCP as transport protocol and binary serialization for data representation. “The binary data format used by the .NET Framework is very much proprietary to Microsoft and hence can only be considered when both the sender and the receiver of the message are on a platform that supports the .NET Framework.” [Burton03]. According to the same author, the binary data format is very compact and efficient.

2.2.7 Remoting-HTTP With Simple Object Access Protocol

Remoting-HTTP uses HTTP as transport protocol and SOAP as protocol for data encoding.

Simple Object Access Protocol (SOAP) is a protocol which is developed by the World Wide Web Consortium. It is an object oriented message passing protocol which uses XML and is therefore human readable. SOAP is a high level protocol which relies on underlying protocols when used as communication protocol.

The design goals for SOAP is simplicity and extensibility and the SOAP framework is developed to be independent of any existing programming model or other implementation specific semantics. [W3C_SOAP03]

2.3 Distributed Component Object Model

2.3.1 Motivation

The main motivation why we have chosen to include DCOM is because it's one of the major distributed technologies available today.

Since it is shipped as a part of every Windows distribution it has an enormous spread around the world. This has spawned a giant repository of third party components.

DCOM was originally proprietary to Microsoft but has been released and are now managed by the independent ActiveX Consortium [ActiveX03].

2.3.2 Overview

Microsoft Distributed COM (DCOM) extends the Component Object Model (COM) and is a technology that enables programmers to create distributed applications. And since DCOM is a seamless evolution of the COM-technology, you can take full advantage of your existing investment in COM-based applications, components, tools and knowledge [MS_DCOM96].

Since DCOM extends COM it works on top of it and uses the functionality provided by COM. So in order to understand DCOM, one has to understand how the COM-technology works first.

Essentially, COM defines how components and their clients interact. The interaction is defined such that the component and the client can connect without any need of an intermediary system component, therefore no overhead whatsoever will be generated.

Of the distributed object middlewares tested, DCOM is the only one that provides binary compatibility with the platform. This means that all DCOM/COM-components are accessed directly on a binary level, without going through any time-consuming marshaling / unmarshaling proxy service. This translates to very fast calls and response times [Rock-Evans98].

To simplify without going to far from the truth; DCOM is a way to distribute COM-components, it takes care of issues such as security settings, remote communication and deployment of components over a network.

Location Independence

When you begin to implement a distributed application on a real network, there are several conflicting design constraints that become apparent. For instance, components that interact more should be "closer" to each other to reduce the amount of overhead. The ratio between component-size and network traffic should be though of when designing the component.

Since there are many constraints concerned with the location of the component DCOM completely hides the location of the component and to match the constraints we only need to

reconfigure the way the components communicate i.e. no changes in the source code or recompilation are needed [MS_DCOM96].

Language Neutrality

DCOM is completely language-independent and virtually any programming language can be used to create COM/DCOM-powered components and applications.

2.3.3 How COM/DCOM Works

Using a COM or DCOM component is exactly like using a local object from the developer's point of view, however the COM/DCOM infrastructure is a bit more complex.

What you do is that you connect to the remote host using the host name or IP and the name or GUID (discussed in the *Interface Description Language* paragraph) of the service. No port is defined since DCOM automatically allocates the port when a connection is made to the SCM (discussed in the *Service Control Manager* paragraph).

A pointer to the remote component is then obtained and through this access to the component can be made.

Working in the Same Process

COM is designed to work so that no overhead is generated. The simplest form of communication is pure version of COM; when components communicate directly through interfaces depicted in figure 2.

Figure 2. Illustrates how components communicates directly in the same process. Originally from the article about DCOM architecture [MS_DCOM96].

Working on the Same Machine in Different Processes

Processes in today's operating systems are shielded from each another. Hence, a client that needs to communicate with a component in another process cannot call the component directly, but rather has to use some form of interprocess communication provided by some underlying functionality such as the operating system.

COM intercepts calls from the client or caller and forwards them to the component resident in another process.

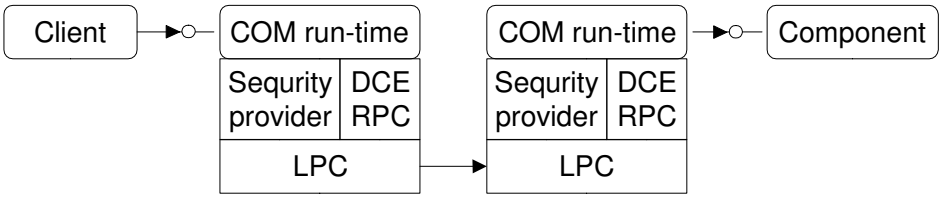

Figure 3. Illustrates how a local call is processed in the COM/DCOM infrastructure. LPC stands for Local Procedure Call.
Originally from the article about DCOM architecture [MS_DCOM96].

Working on Different Machines

As stated before, when clients and components reside on different machines, DCOM falls back on COM in order to gain, among other things, performance.

The COM run-time provides object-oriented services to clients and components and uses RPC and the security provider to generate standard network packets that conform to the DCOM wire-protocol standard.

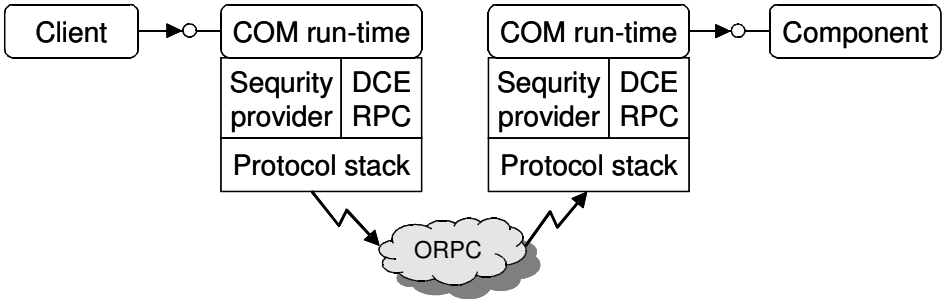

Figure 4. Illustrates the overall DCOM architecture. The interprocess communication is replaced with a DCOM network-protocol.
Originally from the article about DCOM architecture [MS_DCOM96].

2.3.4 The Windows Registry

The Windows Registry is the one single point where all information is located for all DCOM-components. When registering a COM/DCOM-component the information about where the implementation is located and the interfaces that exists is stored in the Windows Registry. The registration, configuration/deployment and maintenance of components are made using external programs since there are many places in the Windows Registry where the necessary information is stored.

2.3.5 Service Control Manager (SCM)

The Service Control Manager (SCM) is the process that is responsible for locating a server and the run the services on it and is a part of the RPC framework. The SCM is in itself a DCOM-component which runs on a specific port (relate to the fact that all other DCOM-components are provided a new port at connection).

When a client requests a service or object to be instantiated, the COM framework contacts the local SCM, the SCM will the lookup the information necessary to connect to the object (which is located in the Windows Registry). If the server that holds the object is on a remote machine, the local SCM will have to contact the remote SCM, which instantiates the object on the server. The final step is to return a pointer to the remote instance of the object. [Grimes97].

2.3.6 Interface Description Language

Objects and their interfaces are defined using an relatively easy-to-use interface description language, which has a C-style syntax and is totally language-neutral. You can also define type libraries, which are extensively used throughout the .NET Framework. The type libraries are integrated into the DLL:s, but in earlier distributions of DCOM, the type library was separately handled.

A type library is essentially a description of the COM elements and are used when registering a COM/DCOM-component in the Windows Registry. Each description of the component is

assigned a GUID (Globally Unique ID) and then registered, that GUID (or the services name) is later used to access the interface through which you access the component.

Unlike the other tested distributed object middlewares DCOM are able to have multiple interfaces. [Rubin99]

2.3.7 Fallback Functionality

COM/DCOM is the only one of the tested distributed object middlewares that has this functionality and it is because COM is so closely integrated to the platform on which it is run and also because it has binary compatibility.

If the COM/DCOM infrastructure detects that the processes are local, i.e. on the same physical machine it falls back to a pure COM (which is a more optimized communication) method. This basically means that it replaces the network protocol with local interprocess communication. Neither the client nor the component is aware that the wire that connects them has just become a little shorter. [Redmond97]

Interprocess communication is by far faster than network communication because it never leaves the process space.

The fallback functionality embodies in the fact that you have to distribute the actual component to all computers that will be using it. The component is then registered together with the component-description and are then setup to run on a specific host instead of the local machine.

2.4 Common Object Request Broker Architecture

2.4.1 Motivation

CORBA is the largest and most widely supported specification for distributed objects since it is developed by Object Managements Group (OMG), which is the largest software organization in the world with over 800 members in the year 2000 [Brose01].

The main reason why we have chosen this technology is because it so widely used and that so many companies support the CORBA specification. Another reason is that it has support for all major programming languages that exist today and especially Java since some of the other distributed object middleware that we are going to test run on the Java platform.

2.4.2 Overview

The acronym CORBA stands for Common Object Request Broker Architecture and it is an open vendor-independent infrastructure that allows applications to communicate with each other over a networks. CORBA works with almost any programming language and operating systems and the communication protocol is standardized so that inter-vendor communication is possible [OMG_CORBA03].

The OMG group is a non-profit organization that is setting standards in the distributed object domain. The OMG group is a vendor-neutral organization that almost all large software companies are members of, including SUN who develops the competing technologies RMI and RMI-IIOP. Also Microsoft has been a member of OMG but according to the member search list at OMG they ain't no more. [OMG_WHATIS03]

2.4.3 How CORBA Works

Distributed Objects in CORBA

When objects communicate with each other they need to know what they can expect from the object they call upon. The services that objects provide are described by a contract in CORBA. The contract serves as an interface between the object and the rest of the environment. An interface in CORBA can be mapped to a class, interface or a struct in programming languages. The contract has two purposes:

- It describes what services the object provides and how to invoke the services in a correct manner.
- It tells the underlying communication infrastructure what the format is on the messages that the object sends and receives. The infrastructure then knows how to translate the data to provide a transparent connection between the object and the client.

The objects also need a handle to enable them to be referenced from clients. This handle is a unique id that each object has. The handle is the same even if the object is moved between servers. [Siegel00]

2.4.4 Object Request Broker

The most important and central part of the CORBA architecture is the ORB (Object Request Broker). The ORB is the service that handles the communication infrastructure. It also handles the relaying of objects across the distributed environment [Mowbray95].

2.4.5 Interface Description Language

The Interface Description Language (IDL) is a language made to define the interfaces of CORBA objects. It is completely language independent and it allows programmers and clients to be assured of type safe invocation of operations. The syntax of IDL resembles the syntax of C++ but it does not contain any programming statements [Brose01].

The IDL specifies the following for an object:

- The operations that an object can perform.
- The input and output parameters that are required for the operations.
- Any exceptions that might be raised during an operation call.

The IDL interface is always separated from the implementation that must be written in a programming language [Siegel00].

The IDL interfaces are compiled into header files and stub programs for the programmers to use directly on the client and server side. IDL could potentially be mapped to any programming language. The most common programming languages are already supported and many are under construction. Usually when buying a CORBA product you get IDL compilers bundled with it.

The generated stub program files is used directly in the programmers code. Method calls look just like local method calls but the method does a transparent call to the remote method via the ORB. On the servers side the generated code skeleton is used. When the ORB gets a request for a specific method it calls the skeleton method, which is implemented by the

programmer. So all the programmer has to do here is to fill in the code in the skeleton class to get the specific functionality [Mowbray95].

Internet Inter-ORB Protocol

Internet Inter-ORB Protocol (IIOP) is the communication protocol that CORBA uses, it uses TCP/IP as the transport protocol and specifies a standard for client and server communication. The interfaces to remote objects are described in a platform-neutral interface definition language (IDL). Mappings from IDL to specific programming languages are implemented, binding the language to CORBA/IIOP, like RMI-IIOP does.

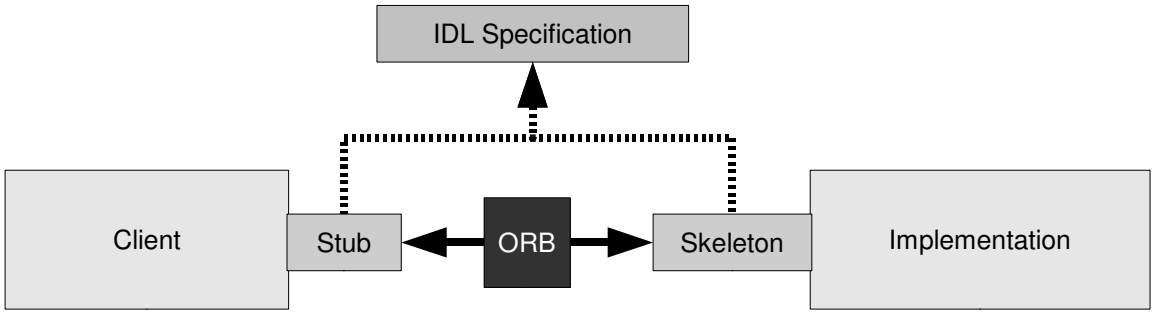

Figure 5. A method call from the client to the implementation.

2.5 Remote Method Invocation

2.5.1 Overview

Remote Method Invocation (RMI) is a technology that enables programmers to create distributed applications in which methods of remote objects can be invoked on the local computer or a remote host. RMI uses object serialization to handle the objects transferred as parameters and return values. The methods are invoked on objects that are referenced either by the RMI naming service, a return value or method parameter [Downing98].

2.5.2 Motivation

The main motivation why we have chosen RMI is because it is a standard part of the Java API and Java is widely used today and one of the largest programming languages in the world. We also feel that it is also interesting to test it against .NET Remoting since they are two competitive technologies.

Another reason is that it uses the Java platform and that is the same platform as we are going to test CORBA with so comparing them against each other will be interesting.

2.5.3 How RMI Works

RMI works almost like any other distributed object middleware. The client connects to the RMI registry to get information on where to locate the requested service. When the service is located the client can connect and call methods via the proxy stub object which makes the calls look like ordinary local method calls.

When you use RMI to develop a distributed application, you follow these general steps:

1. Design and implement the components of your distributed application.
2. Compile sources and generate stubs.
3. Make classes network accessible.
4. Start the application [SUN_RMI03].

2.5.4 Interfaces

Many other distributed technologies uses a secondary language for defining interfaces for the services you aim to implement. Then you have to run a special compiler on the interfaces to generate skeletons and stubs in the language you are going to use. This requires you to know and write the syntax of two different programming languages.

When using RMI however you only need to know Java since that is the language you write your remote object interfaces in. RMI has a special compiler that generates stubs and classes needed for the remote services and clients directly from your Java class files. This compiler is called RMIC [Downing98].

2.5.5 The RMI Registry

The RMI registry is a server that enables clients of RMI services to look up objects that are available for remote method invocation. The registry is used to keep track of the exported remote objects and all the objects in the registry are assigned unique names that are used to identify them. A server that provides remotely accessible objects registers them with the registry and the clients can contact the registry and find out where to find the objects. Registering an object with the registry is called binding in RMI and rebinding if the object is a new instance and replaces an older object. The RMI registry can be started by inside an application but it is normally started as a standalone server that multiple applications can use [Downing98].

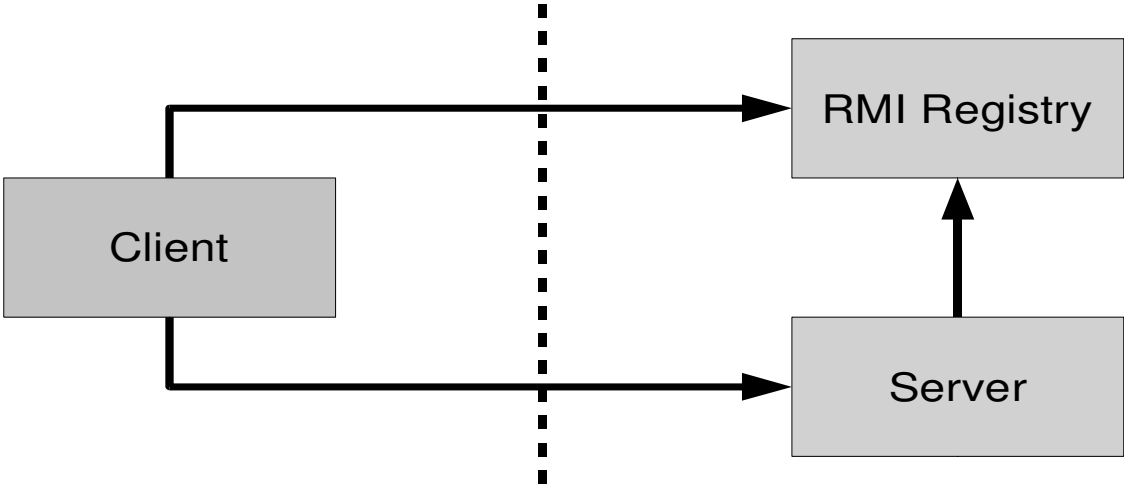

Figure 6. RMI Data flow

2.5.6 RMI-IIOP

RMI-IIOP is a technology that allows a programmer to write fully functional CORBA applications using Java without having to learn CORBA:s IDL language. RMI-IIOP works in a similar way like Java RMI except for the fact that it uses IIOP as its underlying communications protocol and that it uses an ORB to for name services instead of Java's RMI Registry.

We have chosen to test RMI-IIOP because of the fact that it uses the same communication protocol as CORBA and the functionality of Java RMI.

“RMI-IIOP should inter operate with other ORBS that support the CORBA 2.3 specification. It will not inter operate with older ORB:s, because these are unable to handle the IIOP encodings for Objects By Value. This support is needed to send RMI value classes (including strings) over IIOP.” [SUN_RMI-IIOP02].

2.6 Summary

2.6.1 Architecture

The different distributed object middleware that we have tested in this thesis have similar characteristics in their architectures. They all have services, name servers and clients even though the name servers are sometimes started implicitly.

The client part is very similar on all distributed object technologies that we have tested. First a connection to the name server is created then a proxy object to the service is located and instantiated.

The service part is also pretty similar. The three main parts of a service are the interfaces, the implementation and the proxy object. There is an interface which defines which methods the service should provide and there is the implementation of the service which defines what the service should do. The proxy object which contains communication code for making the remote method calls and it also contains marshaling code for serializing parameter and return value data.

2.6.2 Deployment

There are some differences when working with these technologies. One thing that really differs from each distributed object middleware is how the naming service is started. DCOM and Remoting does not start any external program to act as a name server but CORBA and RMI-IIOP uses an ORB and RMI uses an RMI registry. The name servers that CORBA, RMI and RMI-IIOP uses has to be started as separate processes and has their own configurations options.

To start the services from CORBA, RMI, RMI-IIOP you must first make sure that the name server is started and then start your program that registers a service with the name server. You have to define certain parameters to the name servers ex: port and CLASSPATH. The port is needed so that the name server knows on what port it needs to listen to for connections and the CLASSPATH is used to point on the file that holds the service.

To deploy DCOM you have to use external tools to register the component in to the Windows registry. When working with .NET you have to use tools that comes with the .NET framework as sample source code. The component has to be registered on both the client and the server computer and some settings has to be done manually with the DCOM configuration tool.

Remoting does not need any configuration to be started. You can just write your code and start the executable file. You can however do some configuration if you want to but that is either done inside the source code or in an external configuration file.

2.6.3 Run-time

All services of the distributed object middleware we have worked with in this thesis except DCOM are started using a small program that has to be created to bind the service to some kind of naming server which keeps track of all services. DCOM services are registered manually in to the Windows registry by using special tools. The naming service with DCOM is built in to the Windows operating system and launches a DCOM component when it is needed by a client.

3 Experimental setup

3.1 Test environment

The test environment consists of two computers connected through an Ethernet network. One that acts as a server and one that acts as a client. The computers are connected to each other via a 100 Mbit network connection.

3.1.1 Hardware

The client hardware:

CPU:	*Intel Celeron 2.1 GHz*
RAM	*512 MB*
Network Card	*100 Mbit Ethernet card*

The server hardware:

CPU:	*Intel Celeron 500 MHz*
RAM:	*192 MB*
Network Card:	*100 Mbit Ethernet card*

We chose the computer with the highest computing capacity to act as the client so that we could reach maximum CPU load on the server before we reached it on the client. We also used a 100Mbit network to make sure that the network wasn't a bottleneck in the performance tests.

3.1.2 Software

The client and server were both setup in the same way and run the same software.

Microsoft Windows XP

We chose this operating system because it can host both SUN:s Java platform and Microsoft's .NET platform. Windows XP is also, according to usage statistics [OSStatistics03], the second most used operating system over the Internet.

SUN:s Java Platform 1.4.1

This platform hosts our software written for CORBA, RMI and RMI-IIOP distributed object middlewares.

SUN Java Platform can be downloaded for free from www.java.sun.com [20030610].

Microsoft .NET 1.1

This platform hosts our software written for .NET Remoting and DCOM distributed object middlewares.

Microsoft .NET Framework can be downloaded for free from www.microsoft.com [20030610].

Snort 2.0

Snort is network traffic analyzer, which we used for network traffic measurements between the server and the client in the overhead-tests.

Snort is open source and can be downloaded from www.snort.org [20030610].

Cygwin 1.3.22-1

Cygwin is a Linux-like environment for Windows which we used when we automated the performance- and overhead-tests.

Cygwin is open source and can be downloaded from www.cygwin.com [20030610].

3.2 The Test Framework

To make all the tests we wanted to do we had to construct a testing framework. The main reason is that we want a common base for all distributed object middleware, to ensure that we are testing the same things and that all tests have the same premises.

CORBA, RMI-IIOP and RMI are tested on SUN:s Java platform while DCOM and .NET Remoting are tested on Microsoft's .NET platform. Because of the fact that the tests are dependent on these two platforms we had to make the framework platform independent. To be able to achieve this we decided to write the source code in the Java programming language. The Java language is compilable on both SUN:s Java Development Kit and on Microsoft's .NET platform using Visual J#.

There are some differences in the code from Java and J# but we wanted to reuse as much code as possible between the platforms. To achieve this we had the make the coupling as low as possible for loading the different parts of the server and the client. We did this by using singleton factories which look a little bit different between Java and J#. When compiling we copy the source code for the factories in to the correct file depending on which platform the compilation is done. To control which platform we use we have different compilation scripts for Java and for J#. The reason why we have different scripts is also because Java uses the javac compiler and J# uses the vjc compiler.

3.2.1 Requirements of the Framework

When we decided to create the framework we had some requirements that we had decided upon to make the tests as accurate and correct as possible.

- A common test code for all distributed object middleware.
- All of the selected technologies should be able to run on one test framework.
- The framework should be able execute on .NET and the Java platform.

3.2.2 High Resolution Timers in the Test Framework

When measuring performance using time in any computer environment you will need some kind of a timer method. There are plenty of timer methods on most platforms and programming environments. The difference between all of these timers in their resolution. The timer resolution is a measurement in how often the timer method updates its internal time. If a method has a resolution of 100ms and we call that method within a time space of 10ms it would return the same value.

As we use Java as our programming language there is one method available for measuring time and it is java.lang.System.currentTimeMillis(). This method works well in most cases but

for high precision timing it is not so suitable because the time resolution varies on different operating systems. The operating system which we use to perform the tests is Windows XP and currentTimeMillis has a resolution of 10ms with both SUN:s Java and Microsoft's J# and that resolution is not suitable for the measurements we are going to make. [SUN_TIMERS03]

To avoid this problem we have written our own timer method called getTime which uses native calls from the virtual platform to Windows XP. With this method we can achieve a time resolution of 3-4 ms compared to currentTimeMillis's 10 ms.

To verify the resolution of our timing method against currentTimeMillis we use a simple method to capture when the time updates and measure the time from the last update.

```
repeat 3000 times { Time(); } //To make the JIT compile Time();
time = Time();
previous = time;
repeat 5 times {
          // Busy wait until system time changes:
          while(time == previous) { time = Time(); }
          resolution = time - previous;
          previos = time;
}
```

Figure 7. Pseudo-code for measuring time resolution

When using this method for measuring the resolution of the currentTimeMillis and our own timer method we get the following results:

Platform	currentTimeMillis	getTime
Microsoft's J#	10 ms	3-4 ms
SUN's Java	10 ms	3-4 ms

Table 1. Table showing the difference of time resolution.

3.2.3 The Design of the Framework

The test framework is divided up in to a server and a client part. The server and the client are created as command line application and loads distributed object middlewares that are specified through the command line parameters. The distributed object middlewares are compiled in to the programs but they are instantiated through to a common interface to make the coupling as low as possible but without having to deal with loading them dynamically during execution.

The server

The server part of the test framework consists of two major parts. The first part is the distributed object middleware server factory which is responsible for loading the distributed object middleware modules. These modules contain the code for starting a server in the selected distributed object middleware. The second part is the functionality for performing the actual starting of the distributed object middleware.

```
Server server = factory.create(selectedAPI);
server.parseArguments(cmdLineArgs);
server.start();
```

Figure 8. Pseudo code for the server

The distributed object middleware modules consist of a service dispatcher which is the main part of the distributed object middleware module and is responsible for loading and registering of the distributed component. The distributed component contains an implementation of all the test methods that are called remotely by the client.

```
Component comp = new Component();
NameService.bind(ServiceName, comp);
idle();
```

Figure 9. Pseudo code for the service dispatcher

All distributed object middlewares except DCOM are loaded using the server code. The reason why DCOM doesn't use the server to bind itself is because it is registered with external tools into the Windows registry. The Windows registry contains all the information needed for the distributed component to be accessed by clients.

The Client

The client is divided up into three parts. The first part is the client distributed object middleware module factory which creates instances of distributed object middleware modules based on what is selected in the command line arguments. The distributed object middleware modules contains code for calling the naming service and making an instantiation of the proxy object on which the test method calls are made.

The second part is the test factory which creates instances of all the tests that should be executed. These tests are executed using the selected distributed object middleware client module. It is possible to load and execute multiple tests with different parameters but all tests are executed using on one specified distributed object middleware.

The third part is the test executor. The test executer executes the tests in single or multi threaded mode. The single threaded mode is for measuring the execution time for single method calls and for measuring the data sent between the client and the server when a method call is made on a distributed component. The multi threaded mode is for measuring the time it takes for calls when the server has to handle multiple clients.

```
Client client = factory.create(selectedAPI);
client.parseArguments(cmdLineArgs);
foreach(testName in selectedTestNames)
          tests.add(testfactory.create(testName));
if(multithreaded) {
          Threads ts = CreateTestThreads(tests,client);
          foreach(Thread th in ts)
                    th.start();
} else {
          foreach(Test test in tests) {
                    test.performTest(client);
          }
}
```

Figure 10. Pseudo code for the client

Each test class also extends an event handler which makes it possible to write listeners that can receive the results of and progress of each test. We use these listeners to print the result of each test to the command line shell.

3.3 Test Data

All test data has been chosen based on preface-tests that we conducted to get a feel of where the interesting patterns arose, where the limitations where and gain knowledge of how to better enhance/construct our test framework.

The preface-tests and the test cases we formed based on the experimental tests we conducted and from information we got from reading other's work in the field.

The tests included, among other things, overhead and performance measurements for different data types, including complex data and objects. We altered parameters such as string lengths and array sizes to their extremes.

3.3.1 Performance Tests

Primitives

We tested all primitive data types including empty method call.

We expected the empty method call results to be different from the primitive data type test results but it turned out to follow the same pattern and therefore we chose to exclude it in the presentation. We have chosen to only present the results from the tests for byte, double and objects. Byte and double because they are two extremities where byte is 1 byte and double is 8 bytes. Objects however, might be different because distributed object middlewares normally encodes objects differently from primitives.

Complex Data

We tested both arrays and strings with different sizes with all primitive data types.

For the arrays we sent sizes from 0 to 5000 with a delta of 500 items.

For the strings we sent sizes from 0 to 500 with a delta of 100 characters.

Objects

For all distributed object middlewares but CORBA, we sent an instance of java.lang.Object. For CORBA, which can't send an object just by it's own, we sent an instance of java.lang.Object encapsulated in an org.omg.CORBA.Any-object which acts as a container for objects. The reason for us to use org.omg.CORBA.Any is because it seems like CORBA cannot send java.lang.Object. The results might have differed somewhat but since we chose to tests the platforms using their default behavior we couldn't alter this parameter.

3.3.2 Overhead Tests

Primitive Data

We tested all primitive data types including empty method call.

In one of our tests CORBA only uses a 8 bit char-type whilst Java uses a char-type that is 16 bits, this was apparent when we tested the data type char, but to avoid confusion we chose byte as the representative type. Related to the size of char, CORBA requires characters who reside inside the boundaries of the extended ASCII-table, thus making it impossible to send for instance the value 257 using a character. Because of this we only send char values from 65 to 90 or 'A' to 'Z'.

Remoting-HTTP is the only distributed object middleware that is sensitive to the characters actual value in that it cannot send strings that contain characters that isn't registered as a language. For this reason we chose the 'Arabic' – language as a reference-language. This was an arbitrary choice, the only requirement we had on the choice was that it was defined above

the 8 bit ASCII-range (0-255) because this was in relation to the tests were we sent 16 bits character values.

Complex Data

We tested both arrays and strings with different sizes with all primitive data types.
For the arrays we sent sizes from 0 to 100000 with a delta of 500 items.
For the strings we sent sizes from 0 to 25000 with a delta of 500 characters.

Objects

The same as for the performance tests.

3.3.3 Data Type Mappings Between the Architectures

CORBA was the only distributed object middleware where we explicitly specified the interface using CORBA:s own interface description language. Every other distributed object middleware used the corresponding java.lang-types.

Mapping CORBA IDL Type to Java Type

Here follows a mapping of the data types used in the tests were CORBA was involved.

CORBA IDL type	Java type	Remarks
wchar	char	16 bit Unicode character
char	char	8 bit character (ISO latin-1 subset of ASCII) for CORBA and 16 bits for Java
wstring	java.lang.String	Variable length string based on 16 bits Unicode
string	java.lang.String	Variable length string based on 8 bits Unicode for CORBA and 16 bits Unicode for Java
boolean	boolean	TRUE or FALSE
octet	byte	8 bit byte
float	float	32 bit IEEE float
double	double	64 bit IEEE float
short	short	16 bit signed integer
long	int	32 bit signed integer
long long	long	64 bit signed integer
any	org.omg.CORBA.Any	Can contain any type

Table 2. Type mappings for CORBA IDL types to Java types and the size in bits for each type [Matjaz00]

Mapping DCOM type to Java type

Even though we didn't explicitly choose to use the variables defined in table 3 for DCOM, it 's used internally by the COM-runtime. The reason for not using the variables is that when we developed the DCOM-component we used J# on the .NET-platform using .NET-technology which encapsulates such things as native variable-types.

DCOM type	Java type	Remarks
TCHAR	char	16 bit Unicode character
TSTRING	java.lang.String	Variable length string based on 16 bits Unicode

Table 3. Type mappings for DCOM types to Java types and the size in bits for each type.

3.3.4 Motivation of choices in test data

The array size maximum and delta was selected based on what we concluded in the preface-tests. Each of the distributed object middleware:s charts showed that the overhead generated by different array size followed a linear pattern.

4 Results

This section describes the results we got from our tests. Each test case is described individually and the results are the summarized.

All test results were validated through sample/spot tests and from results from others work in the field.

4.1 Measuring Network Traffic

4.1.1 Motivation

The bandwidth on the Internet is limited and the more bytes that are sent, the longer time it will take to transfer it. In conjunction with this, there exists environments where you pay for the amount of bytes transferred across the network and in such settings you will end up paying more if you choose an alternative which sends more bytes per method call than one that sends lesser amount of data. The amount of network traffic can in some cases be equally or more important than the time it takes for a remote method call to be fully executed.

Most of the personal computers today that connect to the Internet is connecting via a modem of some kind. It's a known fact that the bandwidth for modems are lower than for instance broadband or high-speed connections that most servers use. The measurements will provided us with a feel of how feasible each distributed object middleware is towards a usage in non-high-speed environment and on personal computers, rather than servers.

4.1.2 Method

With these tests we aim to measure the total amount of data sent over the network between the client and the server during a single method call using different parameters. To achieve this we use our test framework to invoke a single method call at a time per distributed object middleware. During each call we register the traffic between the client and the server, using a third party network traffic analyzer[1] and measure the total size sent from the client to the server and back.

To prevent other data, such as TCP and IP-headers, from being counted as test data we wrote a special plug-in to Snort which we configured to only register the actual data in every packet. Further, we specified that only the application layer (top layer in Open System Interconnection – OSI [OSISpec03]) data should be registered. Other rules such as host- and source-IP and ports was applied to narrow down the amount of registered data.

Since we had so many tests to perform we decided to automate them as much as we could. Instead of writing yet another application we wrote shell scripts which we executed in Cygwin[2]. The different shell scripts were then feed with different parameters for the different tests.

The parameters that were changed from each test were the following:

- Random content flag
- Data type to send as parameter (for arrays and primitive data types)

1 See *Measuring Network Traffic, Snort 2.0*.
2 See *Test Environment, Software*.

- Length of the contents (for arrays and strings)

The measurement we make are the following:

- The generated network traffic (which are filtered out)

The client setup is discussed in the respective test case. Between each method call we start a new instance of Snort on the client side which logs the transmitted and filtered data on disk.

Snort 2.0

"Snort is an open source network intrusion detection system, capable of performing real-time traffic analysis and packet logging on IP networks. It can perform protocol analysis, content searching/matching..." [Caswell03].

We tried a couple of network traffic sniffers, such as WinDump and Ethreal, before we settled with Snort, the main reason for choosing Snort was because it's open source and we were able to alter the source code by writing a plug-in that only captured the information we needed, i.e. removing information such as TCP- and IP-headers and underlying TCP-protocol control packages.

A specialized plug-in was also written for DCOM, because DCOM opens a unique port for every connection to the service. We were not able to specify a specific port for the DCOM-service on the server, but rather a port-range, so we just adapted the plug-in to only listen to that specific port-range and the problem was solved.

4.1.3 Simple/Primitive Data Types

The primitive data types that we have chosen to measure the overhead for is the ones that have corresponding types for all distributed object middlewares and an empty method call which has no return value and no parameters.

The client applications are set to only send one method call with one parameter of a specified primitive type for the primitive data type tests.

For each of the distributed object middleware:s we also conducted overhead measurements for single-value data to see if there were any optimizations. This was mainly done for the array size tests, but since the single call tests are a subset of the array size tests, it was included.

Here follows pseudo-code which shows how the tests were performed (since we had to start a new server for each distributed object middleware the pseudo-code only shows how a test were performed on a single distributed object middleware);

```
// Array size is not used in the single call tests.
arraySize=[NOT USED IN SINGLE CALL TESTS]

// Data types to test
dataTypes="empty byte char boolean short int long float double"

// The distributed object middleware name and it's parameters,
// such as hosts an ports.
apiStr="[API] [API PARAMETERS]"

// Test both randomized values and the same values.
for randomContent in true | false do
          // Work through all datatypes.
          for dataType in dataTypes do

                    // The test name and it's parameters and the
                       data type.
                    test="[TEST] [TEST PARAMETERS] "dataType" "
                    arraySize" "randomContent

                    // Start listening for data with snort and log
                       to a new file, rules are different for each
                       API, i.e. different ports.
                    snort.exe [SNORT RULES] & > [NEW FILENAME]

                    // Execute our client application.
                    distbench_client apiStr test

                    // When our client is dead, kill snort.
                    kill.exe /name "snort" /signal SIGINT

          done
done
```

Figure 11. Pseudo-code for tests with primitive data types.

Results

Here, we present the results from the overhead measurements for primitive data types and for the empty call method.

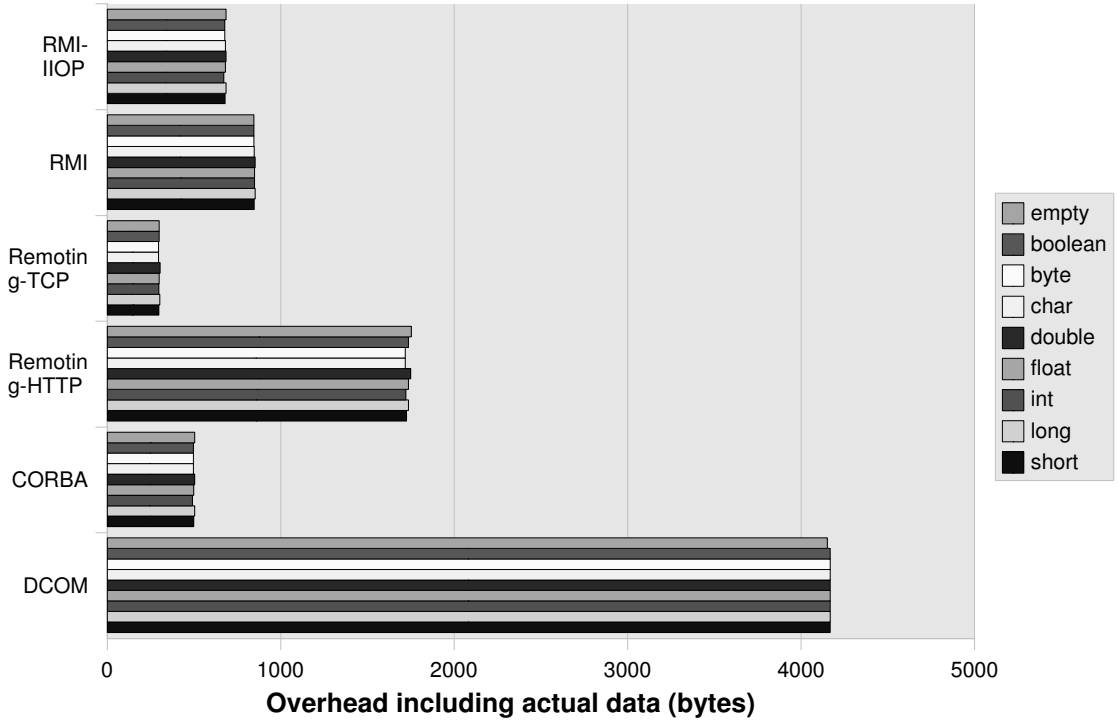

Figure 12. Overhead per data type in each distributed object middleware measured in bytes.

Figure 12 clearly shows that DCOM has very high overhead over the other distributed object middleware:s. From what we've heard and read about DCOM in beforehand, it was a very fast distributed object middleware and therefore we expected results that showed that the overhead would be proportional to the performance of the distributed object middleware. But considering the fact that DCOM[3] is designed for high-speed intranet environments the results are understandable. The reason is that it relies and extends COM, which is non-remotable, and commits itself to base it's protocol on the protocol used by COM.

As can be seen in table 4, DCOM has the same overhead for all primitive data types which might be a somewhat unexpected result. This is due to how DCOM processes the data and how the data is packaged [MS_COMSPEC95], but compared to how much overall overhead DCOM generates the few bytes that might have been won for each primitive type is negligible.

Second runner up in the overall overhead measurement results is Remoting-HTTP, with an average overhead of 1733,4 bytes. This is probably the most understandable result since the transferred data is formatted using SOAP[4] which in essence is human-readable XML.

3 Relate to the information in *Background, Distributed Component Object Model*.
4 See *Background, .NET Remoting, Remoting-HTTP*.

Type	RMI-IIOP	RMI	Remoting-TCP	Remoting-HTTP	CORBA	DCOM
empty	685	845	300	1754	504	4152
boolean	678	846	299	1736	497	4168
byte	678	846	296	1718	497	4168
char	682	847	296	1718	497 / 501[1]	4168
double	685	853	305	1750	504	4168
float	681	849	300	1736	500	4168
short	679	847	298	1725	498	4168
int	673	849	298	1722	492	4168
long	685	853	303	1737	504	4168
string(0 chars)[2]	725	848	298	1738	501 / 500[1]	4184
Average:	685.1	848.3	299.3	1733.4	399.6	4168

Table 4. Overhead measurements including actual data in bytes (same data as in figure 12).
[1] For CORBA we tested both char and wchar, string and wstring (the wide-version is specified last in these cases).
[2] Included string for an overview of how complex data relates to primitive data.

As the table above shows; RMI-IIOP and CORBA:s overhead is similar, this is due to the fact that they share the same underlying protocol, namely *IIOP*. The reason for RMI-IIOP to have a larger overhead is because it also includes Java-specific meta-data on top of the IIOP-protocol.

Of the tested distributed object middleware:s CORBA is the most portable of them all because it's standardized[5]. We expected CORBA:s overhead to be larger because with portability there most often comes an overhead, but the results show that this is not always the case.

We have tested both char and wchar for CORBA, the data type char is only 8 bits and may only contain values from 0-255 or characters that are included in the extended ASCII-table [ASCII02], thus making it impossible to send for instance the value 257. Because of this we only sent char values from 65 to 90 or 'A' to 'Z' for this specific test[6].

The data type wchar, pronounced wide-char is for Unicode-characters and is 16 bits in size, this was used as the reference data type for char in CORBA.

Just as RMI, Remoting-TCP is closely bundled to platform on which it is used, this gives these distributed object middleware:s the “advantage” of the ability to specify non-portable, compact protocols, which clearly shows how small, for instance Remoting-TCP:s overhead is. This also means that it scales well on low-bandwidth environments.

Value-Dependent Overhead

It turns out that the value of character-based variables is crucial to the generated amount of overhead. But for the rest of the tested types all distributed object middleware:s except for Remoting-HTTP are independent of the value of the data sent. Why Remoting-HTTP acts like this can be explained with a small example;

Since Remoting-HTTP is formatted using SOAP, all values such as an integer value is printed out in ASCII-characters, thus making the character representation of the value 1000 larger than that of the value 100, with one character, also meta-data is added about which type

5 See *Background, CORBA*.
6 More about this in the *Test Data* section.

of data is sent. Therefore the overhead for an integer will be larger than a byte not only for the above reason, but because of the fact that the word 'Integer' is larger than the word 'Byte'.

Because of reasons like this, it can't easily be foreseen how much overhead Remoting-HTTP generates just by looking at the data types sent, one also has to look at the value and type of the data.

4.1.4 Array Size Test for Primitive Data Types

For the array size tests, the premises are the same as for the primitive data types tests, with the exception that the client applications are set to different array sizes for each specified primitive type.

Here follows pseudo-code which shows how the tests were performed (since we had to start a new server for each distributed object middleware the pseudo-code only shows how a test were performed on a single distributed object middleware);

```
// From a low array size to a larger.
for arraySize from 0 to 100000 delta 500 do
          // Run the test-script found in figure 11.
          # Run single data type test code here...
done
```

Figure 13. Pseudo-code for tests with array sizes on primitive data types.

As we stated in the Test Data section, the array size maximum and delta was selected based on what we concluded in the preface-tests. Each of the distributed object middleware:s figures showed that the overhead generated by different array size followed a linear pattern. However, there are some interesting patterns on top of the linear pattern which is presented in the next section.

All arrays of primitive types follows the same pattern, except for characters which also follows the patterns described in the string size test section, so we have decided to only present the test results for the tests with arrays containing values of the type byte.

Results

The following figure and table shows a subset (the first ten sizes) of the result data set from the array size tests. This subset is enough to show the patterns found in the result data.

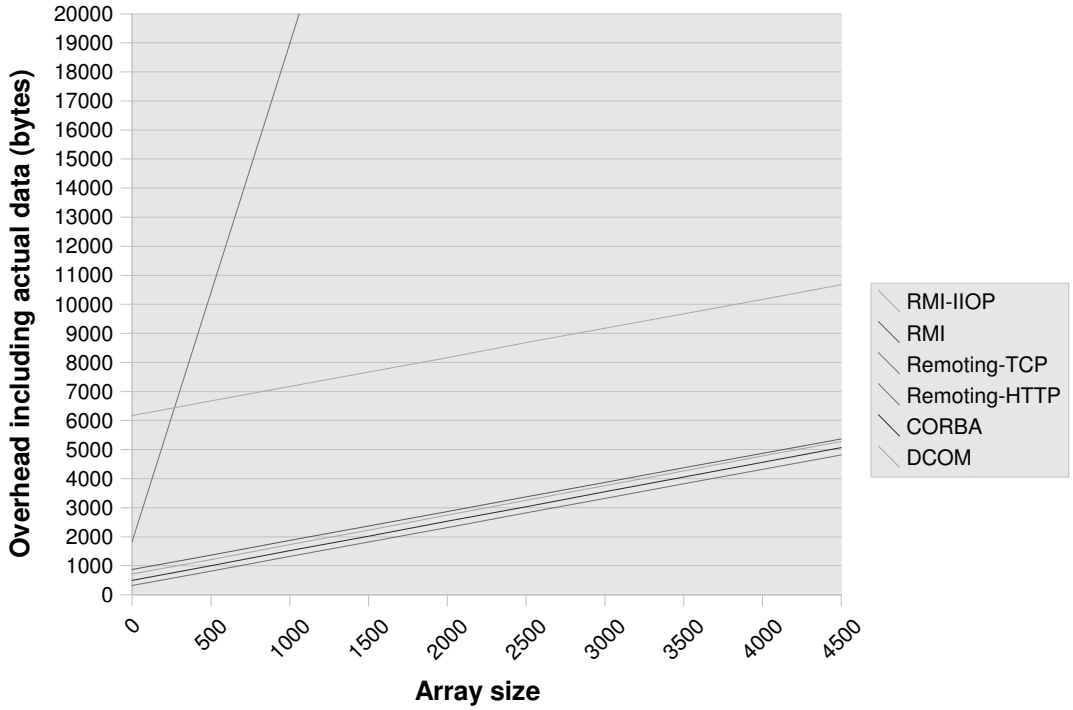

Figure 14. Overhead measurements including the actual data in bytes for arrays of bytes (graphical view).

Figure 14 is scaled down to be able to be able to show the distributed object middlewares relation to each other more accurately.

Array size for array of byte	RMI-IIOP	RMI	Remoting-TCP	Remoting-HTTP	CORBA	DCOM
0	713	869	319	1831	500	6168
500	1213	1369	819	10418	1000	6680
1000	1729	1869	1319	18984	1516	7176
1500	2229	2369	1819	27554	2016	7672
2000	2745	2869	2319	36151	2532	8168
2500	3245	3369	2819	44675	3032	8680
3000	3761	3869	3319	53262	3548	9176
3500	4261	4369	3819	61785	4048	9672
4000	4777	4869	4319	70446	4564	10168
4500	5277	5369	4819	78916	5064	10680
Average:	2995	3119	2569	40402.2	2782	8424

Table 4. Overhead measurements including the actual data in bytes for arrays of bytes.

As we can see in the table above, Remoting-HTTP dramatically increases in overhead size, which shows Remoting-HTTP:s real face. It simply isn't good at scaling with array sizes.

One of the main reasons for this is because Remoting-HTTP:s SOAP formatting of array items looks like this little snippet shows;

```
// A test structure.
struct POINTLIST {
          long cElems;
          [size_is(cElems)] POINT points[];
};

// An SOAP representation of an array of POINTLIST-items.
<t:POINTLIST xmlns:t='uri for POINTLIST'>
          <cElems>3</cElems>
          <points xsd:type='t:POINT[3]'>
                    <POINT><x>3</x><y>4</y></POINT>
                    <POINT><x>7</x><y>5</y></POINT>
                    <POINT><x>1</x><y>9</y></POINT>
          </points>
<t:POINTLIST>
```

Figure 15. A code snippet showing the default syntax for SOAP and arrays [Box03].

As we can see in the snippet above, for each new item in the array, a new <type>value</type> is added, which is an awful way of waisting precious data, but totally in line with the XML-formatting technique. The same phenomena found in the primitive data type tests, where random data versus single value data differs in overhead reflects on the overhead, is found for array sizes but again only for Remoting-HTTP.

Another thing to mention is the fact that the average overhead ratio between for instance DCOM and Remoting-TCP (which were both extremes in the primitive data type test results) have gone from 4168 : 299,3 to 8424 : 2659 or from ≈14 : 1 to ≈3 : 1. In practice this means that the large overhead that DCOM initially had is toned out with the size of the array. The scalability for Remoting-TCP is similar to the other distributed object middlewares except for DCOM and Remoting-HTTP.

Overhead growth

To get a better view of how the growth evolves over the different array sizes, we extract each delta value between all consecutive values in the array list. In other words; how much do the overhead grow for each extra item in the array?

We follow the mathematical formula where the function d is the delta, n is the array size and f is the overhead;

$$d(n) = f(n) - f(n-1) - f(0)$$

Remoting-TCP and RMI have no overhead growth at all, which in practice means that the overhead stays the same independently of the array size.

Because RMI-IIOP and CORBA shares the same protocol, they both show the same pattern, where the delta increases with the array size. The pattern is as follows; For every extra 1000 items in the array, the delta is increased with 16.

Remoting-HTTP has about 16 bytes extra in overhead per item in the array, depending on various reasons related to SOAP-formatting, which is described in the previous section.

The most interesting pattern comes with DCOM, where the overhead delta grows and contracts with this specific pattern;

Array size for array of byte	Delta	Acc. delta
0	n/a	n/a
500	12	12
1000	-4	8
1500	-4	4
2000	-4	0
2500	12	12
3000	-4	8
3500	-4	4
4000	-4	0
4500	12	12
5000	-4	8
5500	-4	4
6000	60	64

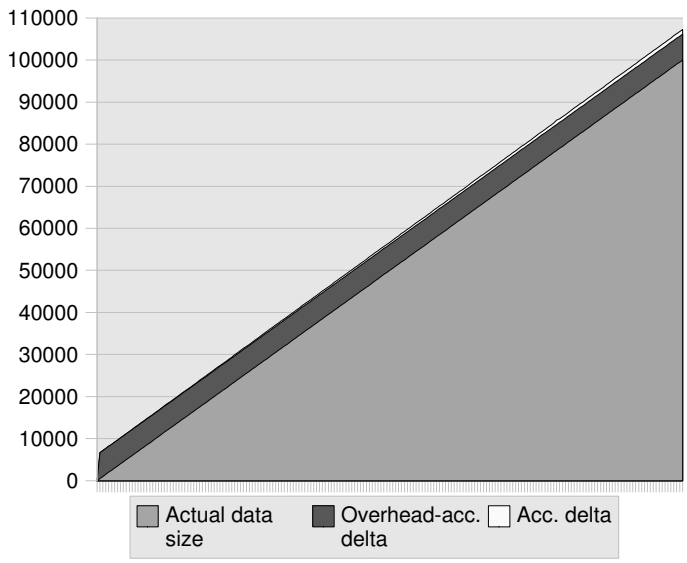

Table 5, figure 16. Overhead delta and accumulated delta for DCOM.

The pattern repeats itself after this, making the accumulated delta grow 60 bytes extra for every extra 6000 items in the array.

4.1.5 String Size Test

For the string size tests, the premises are the same as the other overhead tests. The client application are set to different string sizes in the script for automated tests.

Here follows pseudo-code which shows how the tests were performed (since we had to start a new server for each distributed object middleware the pseudo-code only shows how a test were performed on a single distributed object middleware);

```
// The distributed object middleware name and it's parameters,
// such as hosts an ports.
apiStr="[API] [API PARAMETERS]"

// Test both randomized values and the same values.
for randomContent in true | false do

          // From a low string size to a larger.
          for contentLength from 0 to 25000 delta 500 do

                    // The test name and it's parameters and the
                       data type.
                    test="[TEST] [TEST PARAMETERS] "string" "
                    contentLength" "randomContent

                    // Start listening for data with snort and log
                       to a new file, rules are different for each
                       API, i.e. different ports.
                    snort.exe [SNORT RULES] & > [NEW FILENAME]

                    // Execute our client application.
                    distbench_client apiStr test

                    // When our client is dead, kill snort.
                    kill.exe /name "snort" /signal SIGINT

          done
done
```

Figure17. Pseudo-code for tests with string lengths.

As we stated in the Test Data section, the string size maximum and delta was selected based on what we concluded in the preface-tests. Each of the distributed object middleware:s figures showed that the overhead generated by different string sizes followed a linear pattern, just as for the array sizes.

We conducted two tests; one where we passed 8 bit values and one where we passed 16 bit values to see if any of the distributed object middlewares used any optimization. One should remember that all variables used in the client- and server-applications are 16 bits. Therefore, if the different distributed object middlewares only send 8 bits per character an optimization is done in that case.

Results

The following figure shows how the overhead increases over different string sizes and distributed object middleware:s for 8 bit values. The values we sent were between 65 and 90 or 'A' to 'Z'.

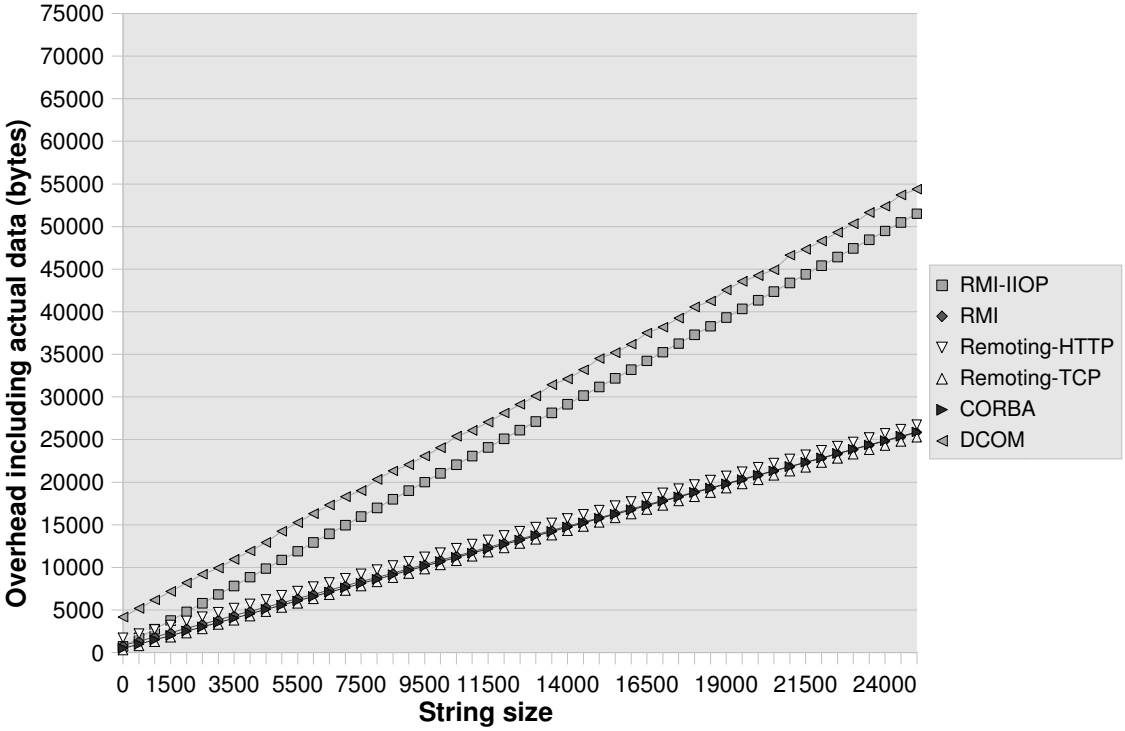

Figure 17. Overhead measurements on different string lengths for 8 bit values. CORBA is using it's 8 bits variable 'char'.

The figure 17 shows that the overhead for RMI-IIOP is about as large as the actual data sent and DCOM is even worse, whilst the other distributed object middlewares have almost no additional overhead what so ever, except for their basic overhead for strings (see table 4 for basic overhead on strings).

The reason for this is because DCOM and RMI-IIOP is always using 16 bits per character, this means that there is no optimization for any of them but it also means that one can easily foresee the size of the transferred data.

RMI-IIOP and CORBA does not act similar here, the reason for this is because RMI-IIOP always use 16 bit Unicode characters and CORBA uses only 8 bits characters (in this case we forced it to do so).

Remoting-HTTP, Remoting-TCP and RMI are all sending only 8 bits per character, meaning they do optimize if possible. This is of course only possible if all characters can be contained inside the boundaries of the extended ASCII-table or the numeric values is between 0 and 255, since they all utilize only one encoding per string. There is no such thing as having different encodings for each individual character, it simply isn't feasible.

In figure 18 we show how the overhead increases over different string sizes and distributed object middleware:s for 16 bit values.

Note that Remoting-HTTP can only send values/characters which are found in any of the languages that are defined in the Unicode-standard. We chose 'Arabic' (numeric range 1536-1791) for this purpose[7].

For the rest of the distributed object middlewares we sent characters in the range from 0 to the maximum value of 16 bits; 65536 (java.lang.Character.MAX_VALUE)[8].

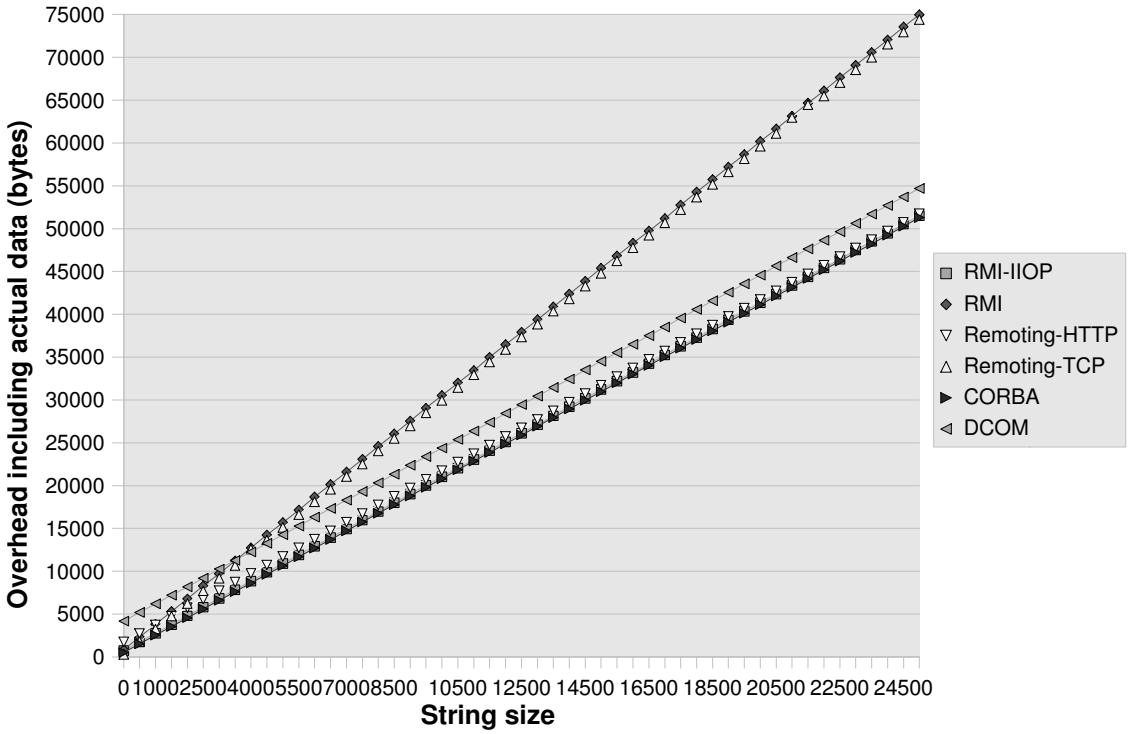

Figure 18. Overhead measurements on different string lengths for 16 bit values.

As we can see in the figure above, all the distributed object middlewares are sending at least 16 bits per character. DCOM and RMI-IIOP lies steadily on the same overhead as in the 8 bits test.

CORBA lies on 16 bits per character as well, because this test is done with CORBA:s reference string data type; wstring. Tests showed that CORBA didn't optimize for 8 bits character values.

Remoting-HTTP has taken a step up and is now using UTF16-encoding, this became apparent when we analyzed the network traffic (the good thing with SOAP formatting is that it's human readable and easy to debug).

The most surprising results comes with Remoting-TCP and RMI. They are now sending 3 bytes per character or 24 bits per character.

According to the Unicode Consortium the new standard for Unicode is as much as 4 bytes per character or UTF32. The Unicode Consortium [UNICODE03] however do not define an UTF24 standard which seems to be the case for Remoting-TCP and RMI.

7 See Test data.
8 See Test data.

We have not been able to find any publicized material regarding UTF24 from Sun Microsystems or Microsoft, but in recent posts on Java forums, and in FAQ lists such as the one on www.Jguru.com, the topic is discussed [JGuru03]. Also, in the official documentation for both Remoting-TCP and RMI it's stated that the respective distributed object middlewares follows the Unicode Consortium's latest release of the Unicode Character Standard (see the latest documentation for java.lang.Character). One should bare in mind that RMI and Remoting are two of the newest distributed object middlewares and therefore it's likely that they are up-to-date.

It's likely that Remoting-HTTP also supports as much as 3 bytes per character but since we chose a language with a fairly low Unicode-range ('Arabic') the UTF16-encoding was used.

Based on the results from the string size tests the estimated functions for each distributed object middleware is as follows;

Distr. Obj. Middleware	8 bit values	16 bit values
RMI-IIOP	$y = 515{,}57 * x + 725$	$y = 515{,}57 * x + 725$
RMI	$y = 0 * x + 848$	$y = 983{,}46 * x + 848$
Remoting-TCP	$y = 0{,}03 * x + 298$	$y = 983{,}84 * x + 298$
Remoting-HTTP	$y = 0{,}03 * x + 1738$	$y = 500{,}02 * x + 1738$
CORBA	$y = 8 * x + 501$	$y = 515{,}59 * x + 500$
DCOM	$y = 510{,}94 * x + 4184$	$y = 510{,}94 * x + 4184$

Table 6. The estimated functions for each distributed object middleware.

4.1.6 Summary

DCOM has by far the highest ground-overhead with an average of 4168 bytes for primitive data types. This is due to the fact that DCOM is designed for high-speed intranet environments.

As we can see human-readable formatting comes with a high price, the ratio between Remoting-HTTP and Remoting-TCP for primitive data types is about ≈6 : 1. This shows very well when we tested the distributed object middlewares with different array sizes. Remoting-HTTP then adds about 17 bytes per extra item in the array.

Remoting-TCP has the lowest ground-overhead with an average of 299,3 bytes for primitive data types which speaks for good scalability in low-bandwidth environments. This is however contradictive to the results for Remoting-TCP in the string size tests, where it doesn't scale well at all. Just as RMI, it uses 3 bytes for every character in a string that has 16 bit values in it.

According to the results, Remoting-HTTP, RMI and Remoting-TCP optimizes for 8 bit values, the rest doesn't.

There are no extra overhead for the first call, even though it takes more time for each distributed object middleware. This is due to the fact that the time consumed goes to things like security checks and negotiations between the server and the client, rather than to additional transfer time and parsing of extra data.

4.2 Time based performance results

4.2.1 Motivation

The motivation for measuring the round trip time (RTT) for method calls and how many calls a server can handle in a certain time space is because it affects the overall execution time and therefore the performance of the application. The number of calls per time unit is important when making a selection based on the server scalability and the round trip time for a certain method is important from the clients perspective.

4.2.2 Method

We used our test framework to execute the tests that we wanted to run. As for the overhead tests, we created shell scripts to make the tests run automatically with different parameter values for the tests each time. The parameters that were changed from each test were the following:

- Number of clients (we used threads to simulate multiple asynchronous clients)
- Data type to send as parameter
- Length of the contents (for arrays and strings)

All tests were done by starting a number of clients with a predefined number of method calls it should make to the server. The number of clients where then increased until it reached the maximum number of threads we had specified for the test.

The different measurements we make are the following:

- The number of calls per millisecond
- The number of milliseconds it takes for a method call to return.

To measure the number of calls per millisecond we divide the total number of calls that where made with the time it took from when the first thread were started to the last thread finished. Since the threads cannot possibly start and stop at the exact same time there is a small amount of time where the threads wait for each other to start and stop. We used the built-in join-functionality for threads to wait for all threads to fully execute. The time consumed cannot be measured so we have specified this as a limitation in our tests[9].

Even though this is a limitation it doesn't affect the results we are after since we measure from the servers point-of-view, namely how many calls can the server handle per time space.

To measure the time it takes for a method call to return we took the time for each thread to execute all its calls divided by the number of calls it made. We then made an average of that calculation over all the threads.

In this section we also calculate the slope of the linear figures showing milliseconds per method call. This value is used to show the difference in how much the distributed object middleware changes for every added client in terms of milliseconds per method call.

The formula for calculating the slope value is:

9 See *Introduction, Limitations*.

$$Slope(k) = \frac{(\Delta Time)}{(\Delta NrOfClients)}$$

Figure 19. The slope formula

4.2.3 Primitives and Objects

In this test we look at performance of the different distributed object middleware when sending primitive data types and simple objects. The data types we use for this tests are the following[10]:

- byte
- double
- object

The reason for not presenting the results from the other data types is because they all follow the same pattern (this is also true for the empty method call which we therefore excluded).

Calls per millisecond when using byte as parameter

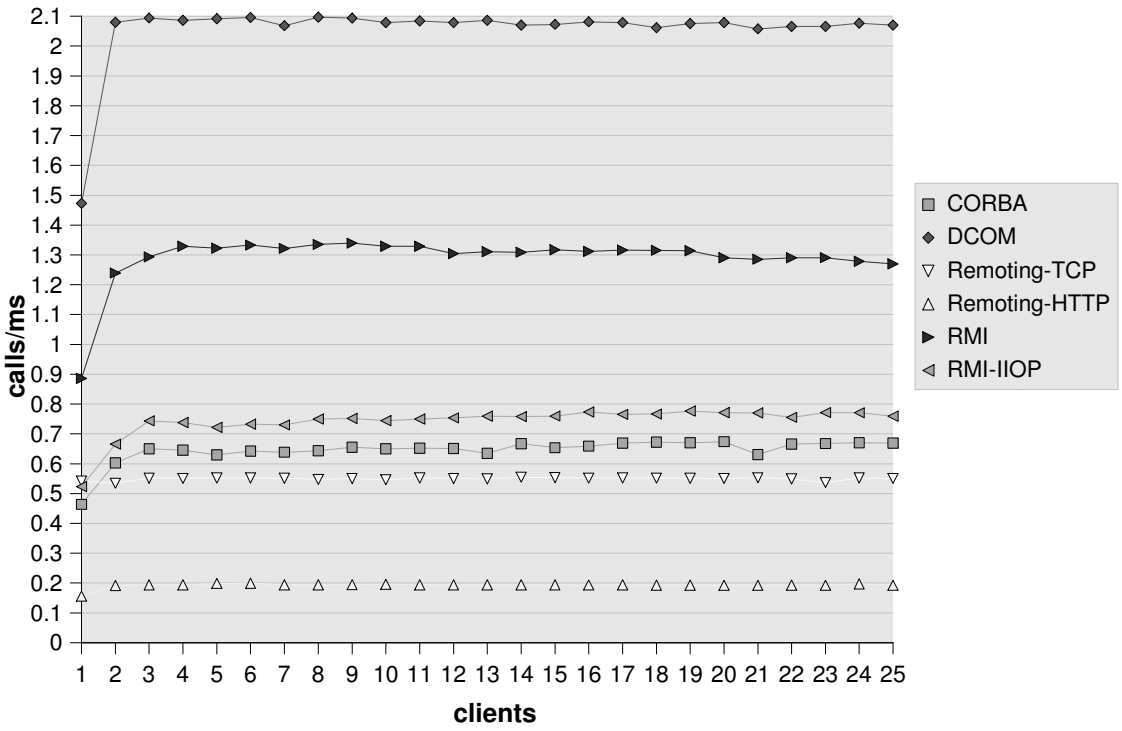

Figure 20. Number of calls per millisecond for making a calls with byte as parameter.

The figure shows clearly that DCOM can handle the most calls per millisecond but it reaches its top performance with just two clients. DCOM performs almost 65% better than the second

10 See *Experimental Setup, Test Data*.

runner up RMI in their peek values. RMI reaches its top performance at about four clients and then degrades slowly by each added client.

CORBA and RMI-IIOP shows almost the same performance pattern as each other. They reach their peek performance at about three clients. This is very natural since they share the same underlying transport protocol. Remoting-TCP has already reached its peek performance with just one client but it performs just below CORBA and RMI-IIOP for every added client.

Remoting-HTTP has the worst performance and can only perform a third of what Remoting-TCP can achieve at the same number of clients.

Milliseconds per call when using byte as parameter

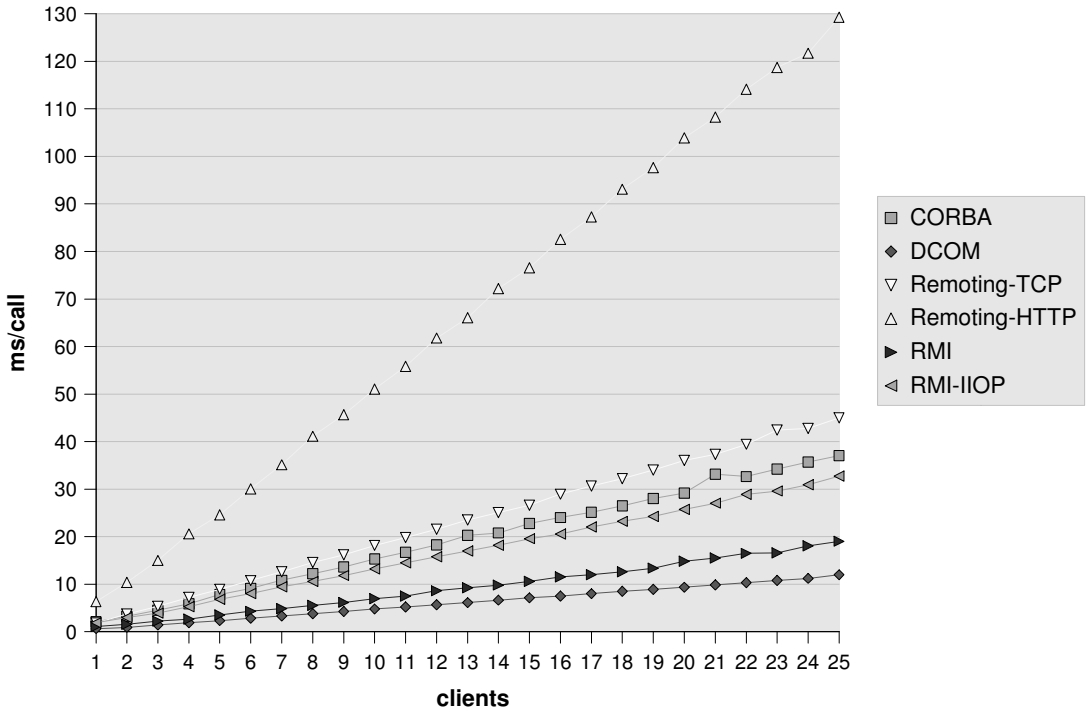

Figure 21. Time in milliseconds per call when making calls with byte as parameter.

Remoting-HTTP has the steepest performance degradation of all the distributed object middleware. Remoting-TCP comes has the second worst performance and is directly followed by CORB and RMI-IIOP.

DCOM has the best performance directly followed by RMI.

Distr. Obj. Mid.	Slope
DCOM	0.47
RMI	0.74
RMI-IIOP	1.28
CORBA	1.47
Remoting-TCP	1.8
Remoting-HTTP	5.16

Table 7. Slope value for each distributed object middleware when using byte as parameter.

The slope values indicates that DCOM has the least degradation over time. RMI comes as second runner up. RMI-IIOP, CORBA and Remoting-TCP comes third, fourth and fifth and Remoting-HTTP is way behind the others.

Calls per millisecond when using double as parameter

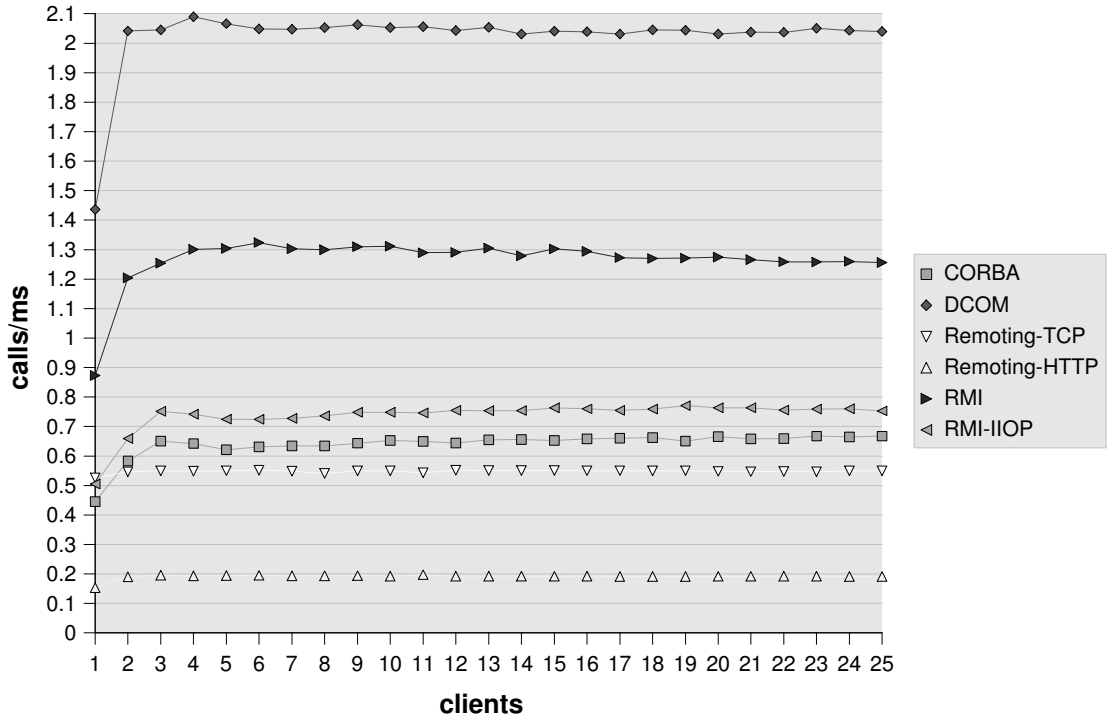

Figure 22. Number of calls per millisecond for making a calls with byte as parameter.

The relation between the different distributed object middleware is very similar to the test where we used byte as parameter. DCOM is clearly outperforms the other distributed object middleware.

In this test DCOM shows that it doesn't reach its peak performance until the fourth client is added to the test unlike in the previous test where it reached its peak after just one client. RMI comes as second runner up followed by Remoting-TCP, CORBA and RMI-IIOP. Remoting-HTTP comes last.

Milliseconds per call when using double as parameter

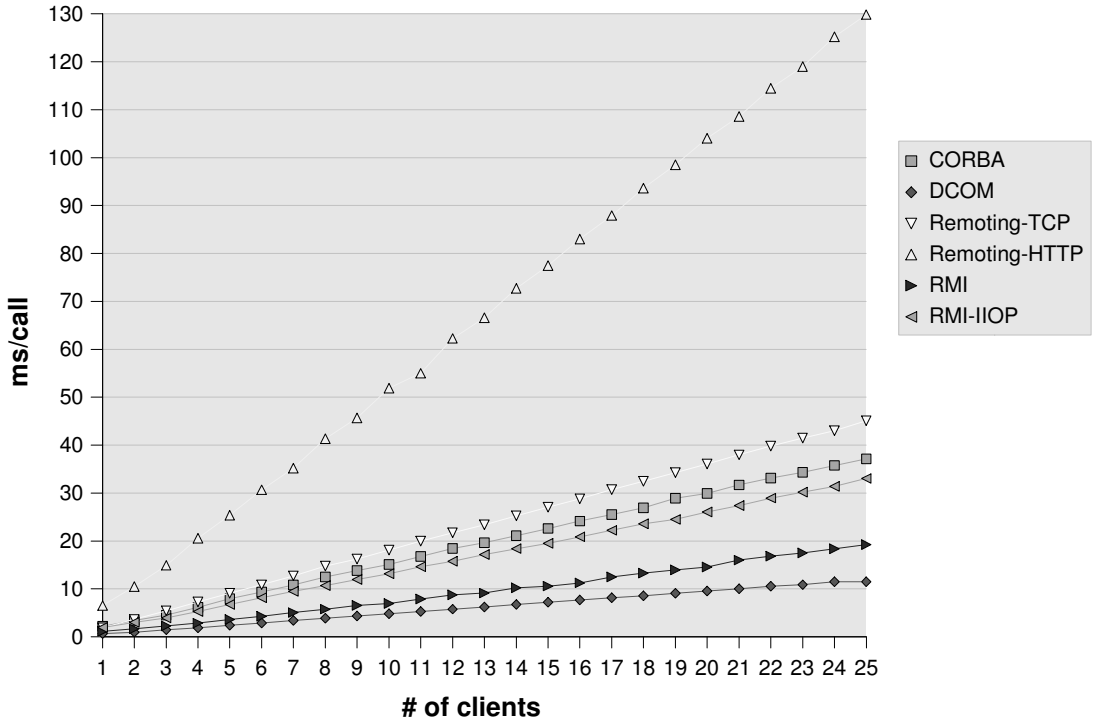

Figure 23. Time in milliseconds per call when making calls with double as parameter.

Here we have the same relations between the distributed object middleware as we had with byte as parameter. The amount of milliseconds per call is slightly less than when using byte as parameter on every distributed every distributed object middleware.

Distr. Obj. Mid.	Slope
DCOM	0.47
RMI	0.76
RMI-IIOP	1.29
CORBA	1.48
Remoting-TCP	1.8
Remoting-HTTP	5.2

Table 8. Slope value for each distributed object middleware when using double as parameter.

The results in table 8 shows the same ranks between the distribute object middleware as when using byte as parameter. The values have increased little which indicates that when using double as parameter the performance will degrade a little bit faster than with byte as parameter.

Calls per millisecond when using object as parameter

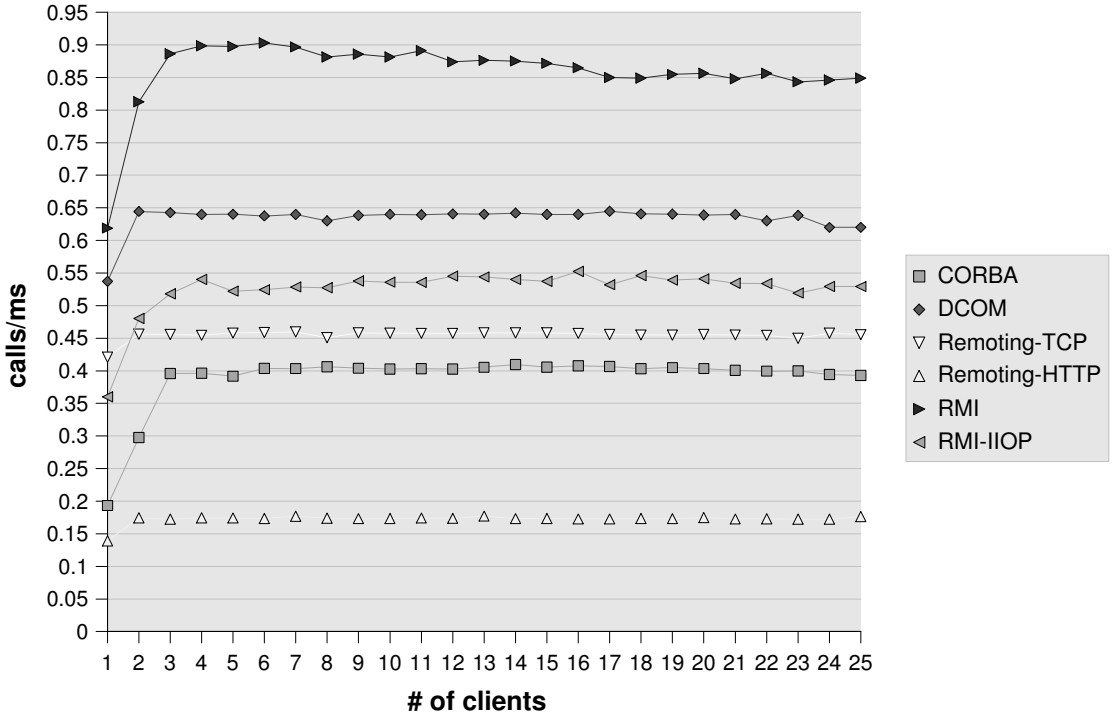

Figure 24. Number of calls per millisecond for making a calls with byte as parameter.

When using object as parameter we can see a bit different result compared to the earlier test results in terms of how they perform against each other. RMI has the best performance in this test followed by the second runner up which is DCOM. DCOM had the best performance when using primitive data types as parameters.

RMI-IIOP performs just below DCOM, but can only handle almost 60% of what RMI can in their peek performances. In the tests with primitive variables we have seen that CORBA is preforming just under RMI-IIOP but when using object as parameter we can see that there is a bigger gap between them. This is probably due to the different measures you have to take with sending objects in CORBA compared to sending objects in RMI-IIOP. CORBA has to have a special container object to hold the real object.

In between CORBA and RMI-IIOP we can see Remoting-TCP which has outperformed CORBA in this test opposed to the test with primitive variables.

Remoting-HTTP comes in last place by far, just like in the previous tests but it has to deal with a lot of XML parsing which is pretty time consuming in comparison to converting binary values.

Milliseconds per call when using object as parameter

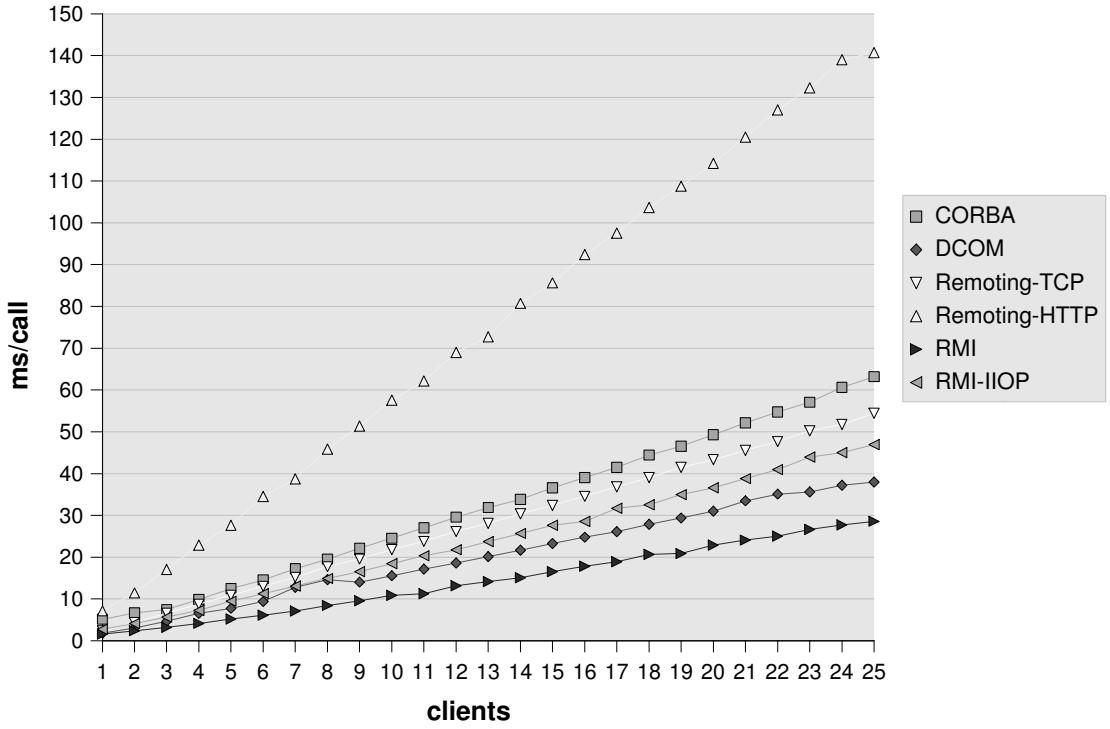

Figure 25. Time in milliseconds per call when making calls with byte as parameter.

The time performance in milliseconds per call shows that same order on which distributed object middleware has the best performance and which has the worst. There isn't so much difference in how much the time increase per added client as it was when primitive data types were used as parameters. The top five distributed object middleware are pretty closely grouped together and Remoting-HTTP is far behind.

Distr. Obj. Mid.	Slope
RMI	1.16
DCOM	1.53
RMI-IIOP	1.85
Remoting-TCP	2.17
CORBA	2.46
Remoting-HTTP	5.73

Table 9. Slope value for each distributed object middleware when using object as parameter.

In this table the ranks have changed from what we saw in the tests with primitive data types. so that RMI is the first followed by DCOM. Remoting-HTTP degrades with almost five times the speed of how RMI degrades. The slope values when using object as parameter is a lot higher than when using byte or double which shows that they will degrade in performance a lot quicker.

Server scalability when using object as parameter

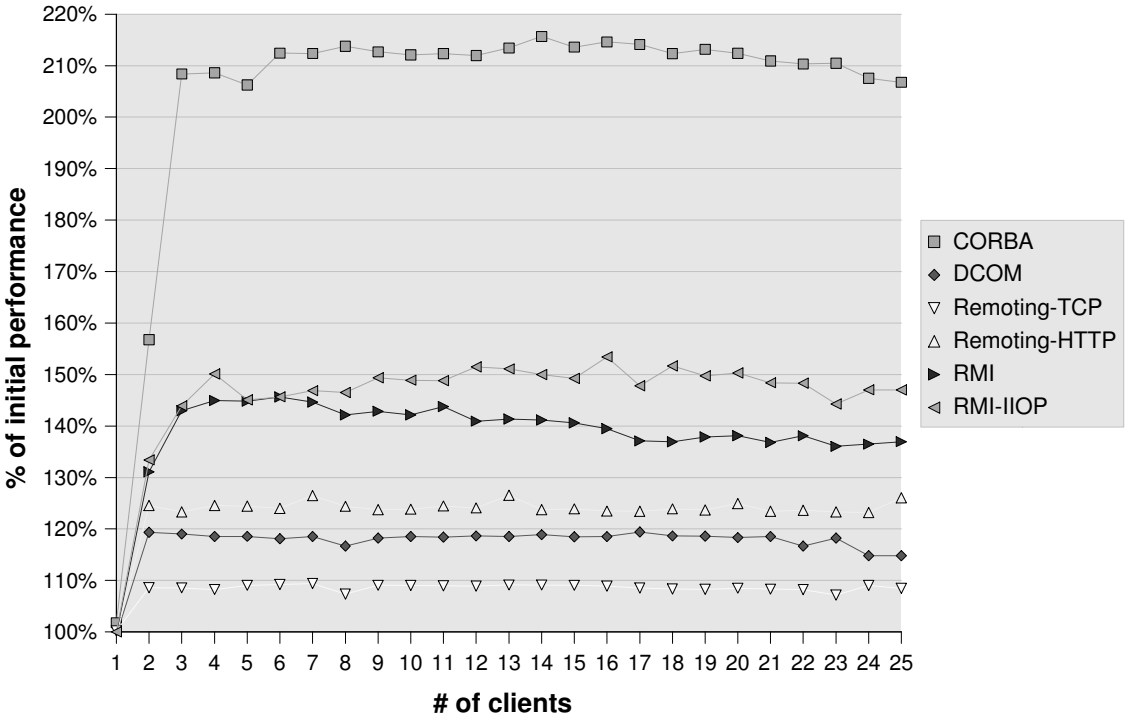

Figure 26. Server scalability when using object as parameter

This figure shows how the different distributed object middleware performs when adding more clients compared to its initial performance when measuring throughput. CORBA which has come in the middle of all the other tests shows that by adding more clients it can really boost its performance.

4.2.4 Strings and Arrays

In this test we look at performance of the different distributed object middleware performance when sending byte arrays and strings in different sizes.

Calls per milliseconds when using byte array as parameter

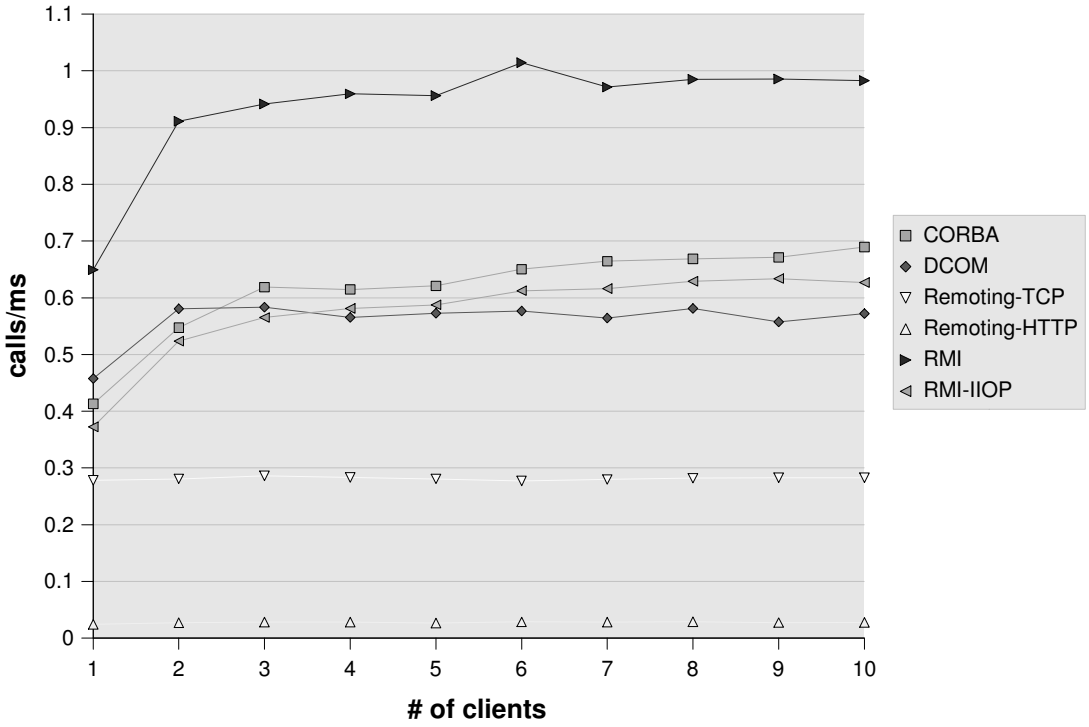

Figure 27. Number of calls per millisecond for making a calls with byte as parameter

To calculate the values for figure 27 we took the average call per millisecond for each number of clients and for each of the array sizes. So for the case when using one client we calculated an average of the results with different array sizes with just one client.

We can clearly see that RMI outperforms the other distributed object middleware with a peek performance which is about 40% better than the second runner up. CORBA surprises by coming as second runner up just followed by RMI-IIOP and DCOM.

DCOM which has had really good performance in the other tests starts of by having the second best performance with two clients but is outperformed by CORBA at three clients and by RMI-IIOP at four clients. Remoting-TCP performs really bad when using byte arrays. It has a maximum performance of only about 30% of what RMI performs at its peek.

Remoting-HTTP:s performance is between 0.02 and 0.03 calls per milliseconds which is really far after the others and thats why it just looks like a flat line in the bottom of the figure 27. This is because of the enormous overhead that Remoting-HTTP sends with arrays.

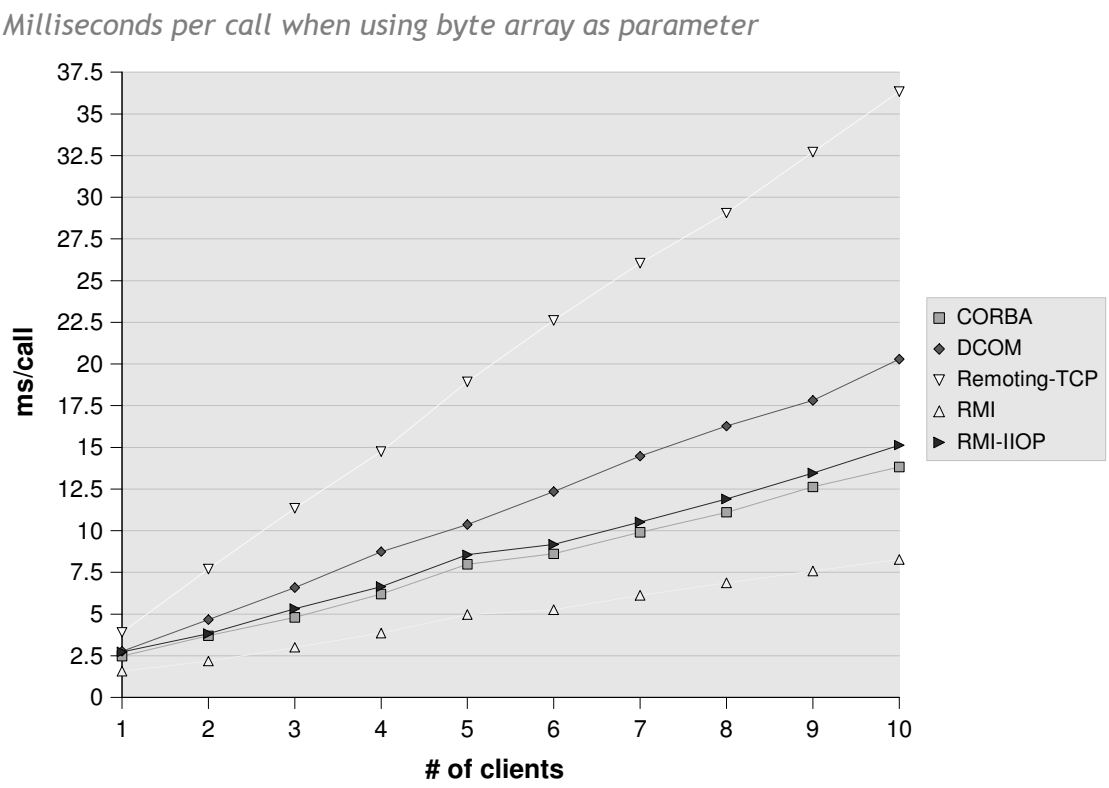

Figure 28. Time in milliseconds per call when making calls with byte as parameter.

We can see the same patten here as with the throughput in figure 27. RMI has the best performance followed by CORBA and RMI-IIOP. DCOM comes after RMI-IIOP and Remoting-TCP is really far behind.

If we look at the RMI, RMI-IIOP and CORBA we can see that the shape of their lines are very similar. There is a distinctive little decrease in performance when using five clients. The same performance decrease can be found in the test with string as parameter. The cause of this similarity is probably because they run on the same virtual platform.

Remoting-HTTP is not in this figure because of its incredibly bad performance in this test. So to make the figure readable we decided not to include it. You can however see the performance of it in this test compared to the other distributed object middleware in table 10.

Distr. Obj. Mid.	Slope
RMI	0.75
CORBA	1.26
RMI-IIOP	1.36
DCOM	1.92
Remoting-TCP	3.59
Remoting-HTTP	98.52

Table 10. Slope value for each distributed object middleware when using byte array as parameter.

Table 10 really shows how badly Remoting-HTTP performs in this test. It has a slope value that is about 131 times worse than RMI.

Calls per milliseconds when using strings with different sizes as parameter

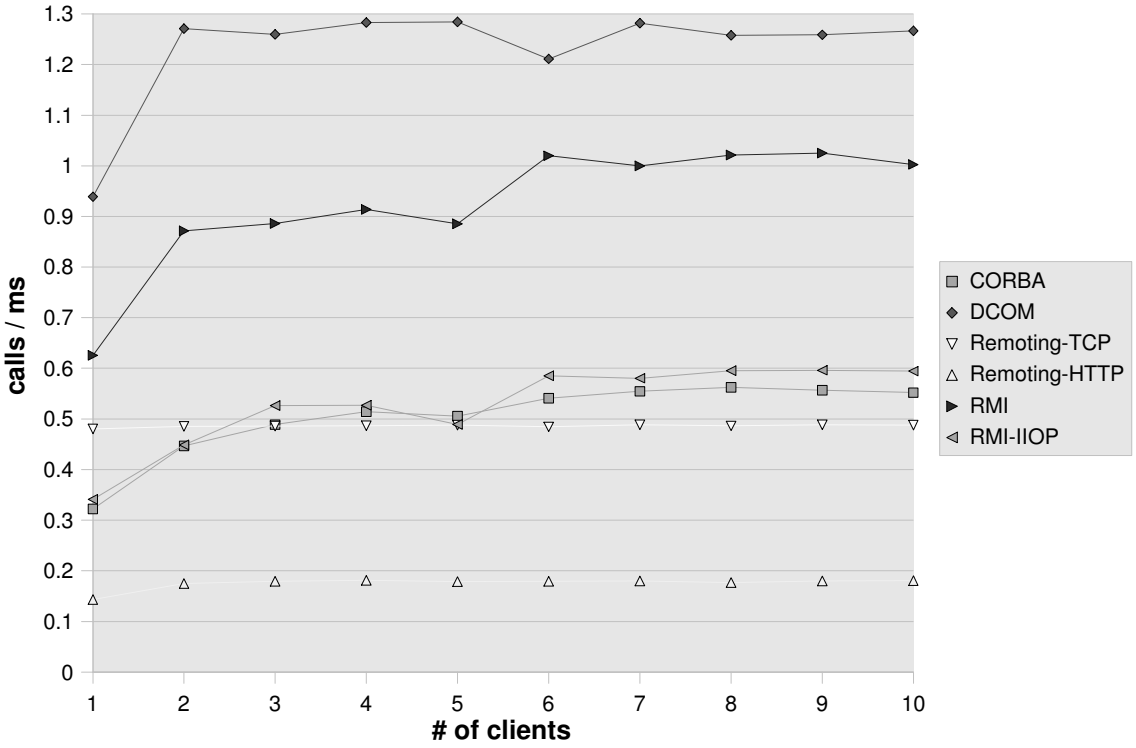

Figure 29. Number of calls per millisecond for making a calls with byte as parameter

As we can see in figure 29 above there are some very interesting patterns such as the way they degrade and when they do degrade.

DCOM is the fastest API when it comes to strings, just as in the case of primitive data types. The pattern for DCOM is rather interesting considering the way it degrades and especially when it degrades. At first we thought this pattern was a coincidence and a measurement error, so we retested it and got the same results.

What we can say about when it degrades is that it is different from the results for RMI, RMI-IIOP and CORBA where the degradation is at five clients only.

We discussed the pattern that RMI, RMI-IIOP and CORBA shows and took a closer look at the test results. It seems like the pattern only exists for string length of one and since figure 27 shows an average of all string sizes per client and distributed object middleware we had to extract the results for RMI, RMI-IIOP and CORBA for string length of one, as can be seen in figure 30.

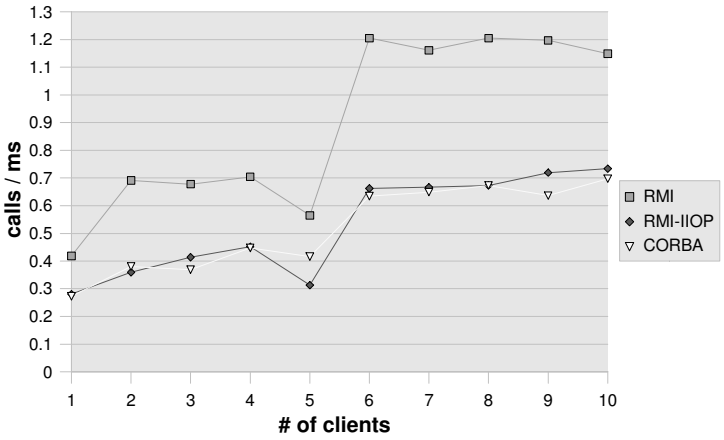

Figure 30. Calls per millisecond for string length 1.

With the results from the overhead tests for strings in mind, our first guess why this was the case was that since we tested both randomized values and singular values on character basis in the strings the impact of the randomized character value would be of highest importance when the string length is 1 or close to 1. Normally the API:s look at the contents of the string and chose the encoding based on the contents. This would mean that an 8 bit value would be encoded using for instance UTF-8 and a 16 bit value would be encoded using for instance UTF-16 or UTF-32. But this can't be the case since RMI-IIOP is known to only send 16 bits per character and the fact that the pattern is the same for three separate tests.

What can we say about the three API:s then? What do they have in common?

They are all run on the same virtual platform, namely Java VM and they all share the same common base-code. Because of this we assume that the pattern is based on the implementation or the common base-code for these API:s. It is not surprising that RMI-IIOP and CORBA is more similar that any other combination of these API:s, because they have more in common since the share the same underlying protocol.

As we can see Remoting-TCP and Remoting-HTTP is very stable and seems to be unaffected by the length of the string. In conjunction with the overhead results for strings and Remoting-TCP we can see that the Remoting-technology is utilizing some form of queuing for requests that makes it that stable. Remoting:s message queuing system is described in detail in the book by Tom Barnaby [Barnaby02].

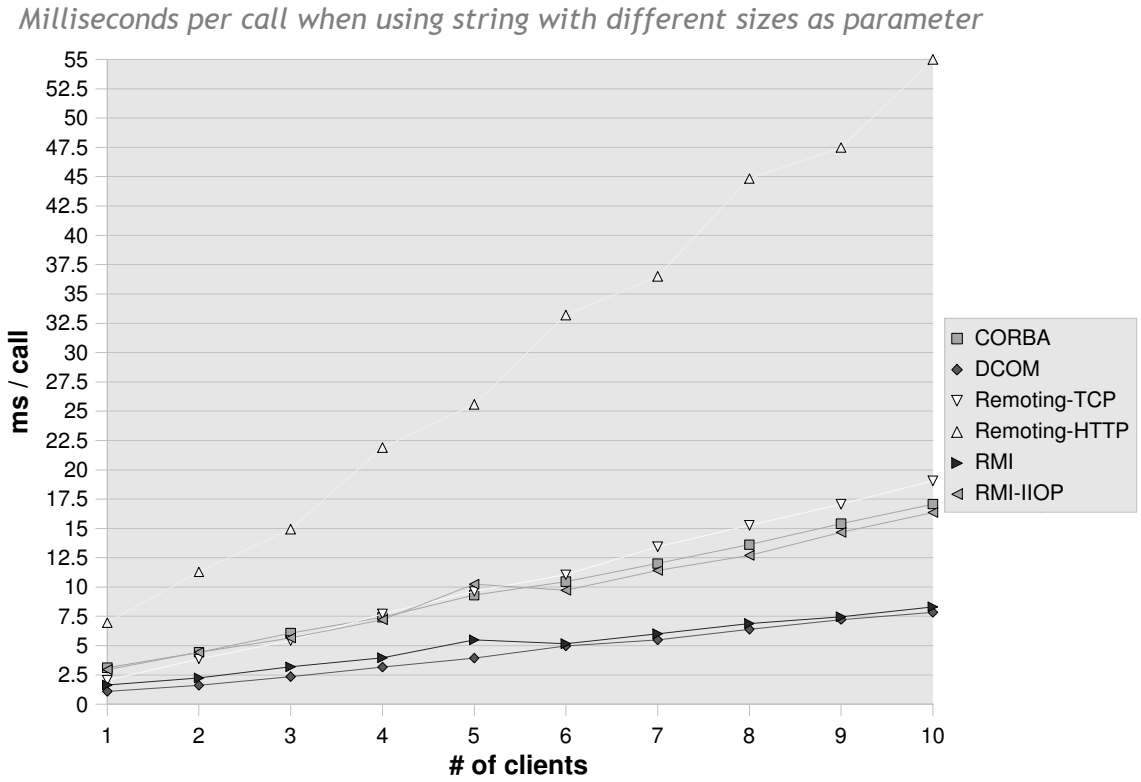

Figure 31. Time in milliseconds per call when making calls with string as parameter.

The results presented in figure 31 above shows that Remoting-TCP is following the same pattern as both CORBA and RMI-IIOP, however it has a slightly higher slope than the other two.

DCOM and RMI:s performance is also similar and this is the first results that show that DCOM and RMI is almost equal in performance.

Distr. Obj. Mid.	Slope
RMI	0.73
CORBA	1.55
RMI-IIOP	1.45
DCOM	0.78
Remoting-TCP	1.9
Remoting-HTTP	5.37

Table 11. Slope value for each distributed object middleware when using string as parameter.

4.2.5 Summary of the throughput tests

Dist. Obj. Mid.	Byte	#	Double	#	String	#	Object	#	Byte Array	#	Average Rank
RMI	1.290	2	1.265	2	0.925	2	0.859	1	0.936	1	1.6
DCOM	2.039	1	2.009	1	1.231	1	0.634	2	0.561	4	1.8
RMI-IIOP	0.742	3	0.737	3	0.528	3	0.525	3	0.575	3	3.0
CORBA	0.645	4	0.640	4	0.504	4	0.390	5	0.616	2	3.8
Remoting-TCP	0.549	5	0.548	5	0.486	5	0.455	4	0.281	5	4.8
Remoting-HTTP	0.193	6	0.191	6	0.175	6	0.172	6	0.028	6	6.0

Table 12. Summary of the throughput tests.

Table 12 above shows the rankings and average values on all the throughput tests we have performed. There is also an average rank column which gives a clue about the distributed object middlewares overall performance.

The table is sorted on the average rank column but this can be misleading in some cases like for example DCOM which has the second best average rank because it is fastest in most of the tests but has really bad performance when sending byte arrays. If you are building an application which heavily relies on doing method calls with byte arrays this technology might be a bas choice.

If we look at the different distributed object middleware from a virtual platform perspective we can see that the technologies that runs on the Java platform outperforms the technologies that runs on the .NET platform. This is probably not so strange because the .NET platform is relatively new compared to the Java platform so it has not had time to mature and become optimized.

5 Discussion

This section will cover thoughts and reflections about the thesis and the work that we have done over the period of writing this thesis.

5.0.1 Working with the Different Distributed Object Middleware:s

What we have found out during this time when working with the thesis and especially when developing the test framework is that the amount of effort you need put in to work with a specific distributed object middleware varies a lot between those we have tested. Even though some distributed object middleware perform very good they can be really hard to work with and cause a lot of trouble for the developer.

The distributed object middleware that was hardest to work with was DCOM. The main reason is because we used it together with the .NET platform and the documentation for doing that where very poorly written and some times missing. The tools that where needed to register a DCOM service with Windows registry was shipped with the .NET platform as sample source code and they where not heavily documented. Since .NET is a relatively new technology there weren't a lot of people that have written about the problem on the Internet so finding help was almost impossible. To get everything to work with DCOM we had to use the trial and error method. Many of the problems we had was with security permissions and component versioning.

CORBA was also pretty hard to work with because interfaces for the services had to be written in a separate language called IDL. But the documentation was good and there where a lot written about it so all problems where resolved pretty quick. CORBA requires that a name server is started. This name server is called ORB.

Remoting however which is an integrated part of .NET was very easy to work with both in implementation and deployment. The documentation for using it was very extensive and there where many tutorials and guides on the Internet. One of the best things with Remoting was that we didn't have to start any name server to use it which made it really easy to deploy the service. As with DCOM we also had some problems with versioning.

RMI and RMI-IIOP was also relatively easy to work with. Not as easy as Remoting but there where no major problems and the documentation was pretty extensive. The deployment of these two was dependent on a name server that had to be started explicitly. RMI uses the RMI registry and RMI-IIOP uses an ORB which is the same as what CORBA uses.

One of the main challenges when working with these technologies was to bind them to our test framework and that also turned out to be a little problem. They all worked basically the same way but we had to do some hard coding to make them all fit in. Most of the problems we had with integrating them into the framework where that we had to make the framework compilable on both the Java and the .NET platform. On the .NET platform we used the J# language which is similar to the Java language but has some small differences. This forced us to make conditional build scripts which acted a little bit different on each of the two platforms.

6 Related Work

When writing this thesis we stumbled upon a lot of similar reports where different distributed object middleware were tested in many different ways. Most of the reports often only tests two or three different technologies and there are some differences in what methods they use.

Java 2 distributed object middleware performance analysis and optimizations

This report deals with the distributed object middleware technologies that comes with the Java 2 platform and their performance. It also looks in to some optimizations that can be done to make them perform better. The optimization techniques they present in this report really improves the performance of CORBA and RMI-IIOP. [Matjaz00]

RMI, RMI-IIOP and IDL Comparison

This is a report which compares the performance of RMI , RMI-IIOP and CORBA. They compare the technologies on SUN:s Java 2 platform and does a comparison of the performance based on two different versions of the Java 2 platform. The results they get shows that CORBA performed worse than RMI in the earlier versions but had similar performance in the latest version that was available at the time of writing for that report. [Rozman01]

Comparison of Middleware Technologies – CORBA, RMI & COM/DCOM

This thesis aims to compare the middleware technologies from an architectural perspective. The thesis also examines aspects like ease of programming with the different technologies. Their results shows differences in their architecture and applications. [Patil01]

7 Future Work

The work in this thesis can be extended in a number of ways to make it more complete and to cover a larger domain. The work can be extended by testing more distribute object middleware or other means of communicating between applications like remote procedure calls or ad hock protocols. Other methods can also be used to test the distributed object middleware in different ways.

The thesis can also be extended with more background information on each distributed object middleware and how easy or hard they are to work with.

7.1 Other Distributed Object Middleware & RPC Middleware

The are a lot of other technologies to test in the same domain. They are all different in many ways but they all share the ability to make method calls on remote computers while making it look like it is a an ordinary local method call.

7.1.1 Web Services

Web services is a way of making object oriented remote procedure calls on service hosted on web servers. Web services uses SOAP as encoding and HTTP as transport protocol. Web services has many different implementations and is available for many platforms but should be inter operable. The world wide web consortium is the organization that develops the web service specification and it is an organization which has many large companies as members. To read more about web services you can follow the link below.

Web services at W3C: http://www.w3c.org/2002/ws

7.1.2 XML-RPC

XML-RPC is a remote procedure call technology which uses XML as encoding and HTTP as transport protocol just like Remoting-HTTP which uses SOAP as encoding and HTTP as transport protocol. The main difference between the encoding that XML-RPC uses and SOAP is that it is designed to be as simple as possible and that it is not object oriented. It supports simple data types and complex data structures as parameters and return values. It has implementations in many programming languages and can be executed on many platforms. To read more see the link below:

XML-RPC:s Homepage: http://www.xmlrpc.com

8 Conclusion

The goal of this thesis is to make a performance comparison between different distributed object middleware technologies. The distributed technologies that we decided to test were:

- CORBA
- Remoting-TCP
- Remoting-HTTP
- RMI
- RMI-IIOP
- DCOM

We wanted to see how they performed against each other in terms of throughput, round trip time and network traffic load. To make all these tests we had to create a common test framework where all distributed object middleware could be tested. The test framework had to be able to execute on SUN:s Java platform and on Microsoft's .NET platform to be able to test the technologies we had chosen.

To measure the throughput we measured how many method calls a server could handler per millisecond with different parameter types.

To measure round trip time we took the average time it took for a method to return when using a defined number of clients and with the use of different kinds of parameter types.

To test the network traffic load for each method call we made a single method call with each distributed object middleware and measured the total amount of bytes sent from and to the client during the call. To measure this we used a third party tool called Snort which captured network traffic and statistics.

Performance in Terms of Throughput and Round Trip Time

DCOM was the distributed object middleware that showed the best results in our throughput and round trip time tests when using primitive data types as parameters. RMI was the fastest just before DCOM when using simple objects as parameters and second runner up after DCOM when using primitive data types.

RMI-IIOP and CORBA had pretty similar performance but RMI-IIOP was slightly better in most of the tests. The reason for them being so similar is mostly because they use the same transport protocol IIOP.

Remoting-TCP had performance that where slightly worse than RMI-IIOP and CORBA overall but was a bit faster than CORBA when using object as parameter. Remoting-HTTP however was the worst performer in all tests and that wasn't unexpected since it uses XML with SOAP as encoding and HTTP as transport protocol which are both text based.

When looking at performance of the distributed object middleware from a virtual platform perspective we found that the distributed object middleware that came with SUN:s Java platform where generally faster than the ones that came with Microsoft's .NET framework

except for DCOM. This is probably due to the fact that the Java platform has been around for many more years than .NET.

Network Traffic

The results showed that DCOM by far has the highest ground-overhead with an average of 4168 bytes for primitive data types. This is due to the fact that DCOM is designed for high-speed intranet environments.

The results also showed that Remoting-HTTP:s SOAP-formatting comes with a high price, the ratio between Remoting-HTTP and for instance Remoting-TCP for primitive data types is about ≈6 : 1. For arrays Remoting-HTTP adds about 17 bytes per extra item in the array which is very much compared to the other distributed object middlewares.

Remoting-TCP has the lowest ground-overhead with an average of 299,3 bytes for primitive data types which speaks for good scalability in low-bandwidth environments. These results are however contradictive to the results for Remoting-TCP in the string size tests, where it doesn't scale well at all. Just as RMI, it uses 3 bytes for every character in a string that has 16 bit values in it. For 8 bit values RMI, Remoting-TCP and Remoting-HTTP optimizes and only sends 8 bits per character.

Choosing a Distributed Object Middleware based on Network Environment

In a heterogeneous network environment we would recommend CORBA or RMI-IIOP because their performance is in the middle of the other technologies and they are compatible with many other CORBA implementations in various languages and on many different platforms. However if you have the Java platform installed on your computers you might as well use RMI since it was faster than both CORBA and RMI-IIOP and a little bit easier to deploy.

In a homogeneous network environment with Microsoft's Windows operating system installed we think that a DCOM is a good choice when it comes to performance though the configuration and development time might be a little longer.

9 References

[**ActiveX03**] "ActiveX Consortium", n/a, 2003 * http://www.activex.org

[**ASCII02**] "Extended ASCII", Webopedia.com, 2003 * http://www.webopedia.com/TERM/E/extended_ASCII.html

[**Barnaby02**] "Distributed .NET Programming in C#", Tom Barnaby, Apress, 2002 [ISBN: 1-590-59039-2]

[**Box03**] "A Young Persons Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages", Don Box, 2003 * http://msdn.microsoft.com/msdnmag/issues/0300/soap/default.aspx

[**Brose01**] "Java Programming with CORBA", Gerald Brose, Andreas Vogel Keith Duddy, OMG Press, 2001 [ISBN: 0-471-37681-7]

[**Burton03**] "Introduction to Remoting", Kevin Burton, samspublishing.com, 2003 * http://www.samspublishing.com/dotnet_home/authors/burton/articles.asp

[**Caswell03**] "Snort 2.0: Intrusion Detection", Caswell et al., Syngress, 2003 [ISBN: 1-931-83674-4]

[**Downing98**] "Java RMI : Remote Method Invocation", Troy Bryan Downing, IDG Books Worldwide, Inc., 1998 [ISBN: 0-7645-8043-4]

[**Grimes97**] "Professional DCOM Programming", Dr. Richard Grimes, Wrox Press, 1997 [ISBN: 1-861000-60-x]

[**JGuru03**] "Serialization FAQ From jGuru", n/a, 2003 * http://www.jguru.com/faq/printablefaq.jsp?topic=Serialization

[**Matjaz00**] "Java 2 Distributed Object Middleware Performance Analysis and Optimization", Matjaz et al., n/a, 2000 * http://lisa.uni-mb.si/~juric/j2midware.pdf

[**McLean03**] "Microsoft .NET Remoting", McLean et al., Microsoft Press, 2003 [ISBN: 0-7356-1778-3]

[**Mowbray95**] "The Essential CORBA", Thomas Movbray, Ron Zahavi, OMG Press , 1995 [ISBN: 0-471-10611-9]

[MS_COMSpec95] "The Component Object Model Specification", Microsoft Corporation, 1995 * http://download.microsoft.com/download/6/b/e/6be6ffaa-6c65-48a7-a172-103469a33a1a/COM1598B.ZIP

[MS_DCOM96] "DCOM Technical Overview", Microsoft Corporation, 1996 * http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp

[Obermeyer03] "Microsoft .NET Remoting: A Technical Overview", Piet Obermeyer and Jonathan Hawkins, Microsoft Corporation, 2001 * http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp

[OMG_CORBA03] "CORBA® BASICS", n/a, 2003 * http://www.omg.org/gettingstarted/corbafaq.htm

[OMG_WHATIS03] "What is the OMG?", n/a, 2003 http://www.omg.org/memberservices/index.htm

[OSISpec03] "International Organization for Standardization", n/a, 2003 * http://www.iso.org

[OSStatistics03] "Operating System Usage Statistics", n/a, 2003 * http://www.epistemelinks.com/Info/OpSys.aspx

[Patil01] "Comparison of Middleware Technologies", Patil et al., n/a, 2001 * http://www.egr.msu.edu/~patilabh/Final_Report.pdf

[Rammer02] "Advanced .NET Remoting", Ingo Rammer, APress, 2002 [ISBN: 1-5905-9025-2]

[Redmond97] "DCOM: Microsoft Distributed Component Object Model", Frank E. Redmond III, IDG Books Worldwide, 1997 [ISBN: 0-7645-8044-2]

[Rock-Evans98] "DCOM Explained", Rosemary Rock-Evans, Digital Press, 2003 [ISBN: 1-55558-216-8]

[Rozman01] "RMI, RMI-IIOP and IDL Comparison", Rozman et al., n/a, 2001 * http://lisa.uni-mb.si/~juric/jr3.pdf

[Rubin99] "Understanding DCOM", Rubin et al., Prentice-Hall, 1999 [ISBN: 0-13-095966-9]

[Siegel00] "CORBA 3: Fundamentals and Programming", Jon Siegel, OMG Press, 2000 [ISBN: 0-471-29518-3]

[Stallings98] "Operating Systems", William Stallings, Prentice-Hall Inc., 1998 [ISBN:0-13-917998-4]

[SUN_RMI03] "RMI tutorial", n/a, 2003 * http://java.sun.com/docs/books/tutorial/rmi/index.html

[SUN_RMI-IIOP02] "RMI-IIOP Programmers Guide", Sun Microsystems, Inc., 2002 * http://java.sun.com/j2se/1.4.1/docs/guide/rmi-iiop/rmi_iiop_pg.html

[SUN_TIMERS03] "My kingdom for a good timer", Vladimir Roubtsov, n/a, 2003 *
http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html

[Tallamraju03] "ASP 101: .NET Buzzword Reality", Jayram Tallamraju, n/a, 2003 *
http://www.asp101.com/articles/jayram/buzzword/default.asp

[Unicode03] "Unicode Consortium", n/a, 2003 *
http://www.unicode.org

[W3C_SOAP03] "W3C: World Wide Web Consortium", n/a, 2003 *
http://www.w3c.org

* All web pages have been successfully tested prior to the publication of this thesis 2003-06-10.

ABBREVIATIONS

CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
HTTP	HyperText Transfer Protocol
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
TCP	Transmittion Control Protocol
XML	Extensible Meta Language