



MinCor.NET

Version: 2.4

Middsol GmbH
mincor@middsol.de

Contents

1	Introduction	3
2	Welcome to CORBA	4
3	Welcome to MinCor.NET	10
4	Installing MinCor.NET	11
5	Open Issues.....	11
6	A Simple CORBA Application	12
7	IDL-to-C# -Mapping	17
8	Tools	20
9	Runtime Configuration	22
10	Client Connection Model.....	24
11	Sources and References	25

1 Introduction

Today, there is an increasing market for mobile applications build on top of a great variety of different platforms, and Mirosoft's Windows CE is holding an also increasing share of this market. The basic technology for programming Windows CE applications is the popular Microsoft .Net Compact Framework.

Unfortunately, the .Net Compact Framework lacks the standard .Net Remoting feature which is part of the .Net Framework for desktop and server platforms. **Middsol's MinCor.NET** fills this gap by providing standard CORBA inter-process communication for the .Net Compact Framework. Beside this, the whole world of C++ and Java embedded Orbs, as well as advanced, tightly coupled client-server architectures (instead of the more loosely coupled Web Service approach) is opened to the .Net programmer, using MinCor.NET.

Therefore, MinCor.NET integrates the most important component architectures competing within the software development market, enabling the easy implementation of mobile components for each of it:

- **Microsoft .Net**, the new standard development platform on top of all the Microsoft Windows operating systems, dominating the client desktop worldwide.
- **CORBA**, as a well established, non proprietary, independent integration platform. CORBA has a long track record for integrating legacy systems.
- **Enterprise Java Beans (EJB)**, Suns standard for server side Java development.

This manual will give a brief overview of the CORBA platform and how to build CORBA clients as well as servers on top of Microsofts .Net Compact Framework using C# and MinCor.NET. For a complete reference, please, refer to the file **MinCor.NET.chm** which is a compiled HTML help documentation. When you open the file **index.html** in the **MinCor.NET root folder** you will find a guide through other parts of the documentation and you will also find a link to the **tutorials** which we highly recommend to follow. If you are familiar with CORBA in general, you may skip the following section and start reading section 3.

2 Welcome to CORBA

The CORBA Architecture

The name CORBA is (like a lot of terms within IT business) an acronym, standing for

Common Object Request Broker Architecture

It is actually not a software, but a mere specification defined by the **Object Management Group (OMG)** with around 700 members – including Microsoft – the worldwide biggest consortium within the software industry. The current version (June 2004) is CORBA v3.0.3.

CORBA is a comprehensive, non-proprietary reference model for the development of distributed applications with the concept of the **Object Request Broker (ORB)** at its core.. Inside a CORBA application, the ORB plays the role of a software bus connecting several modules whose interfaces are defined using a standardized **Interface Definition Language (IDL)**. The IDL, although strongly resembling C++ , is nevertheless a language of its own to describe interfaces (i.e. types and signatures) independently of any specific implementation language. The clue to any concrete programming language is provided by using an equally standardized mapping of the IDL into the data types and signatures of the target language.

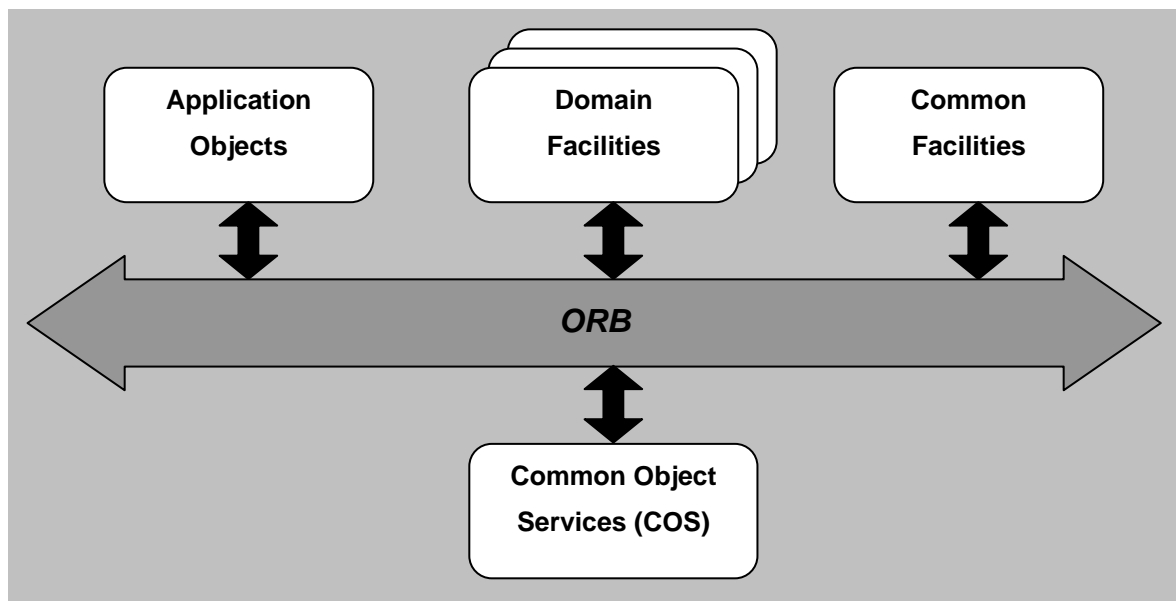
Besides the definition of the IDL and its mapping into several target languages (currently, there are standardized mappings into C, C++, Java, Smalltalk, Cobol, CommonLisp, Python and CorbaScript available, with a C# mapping pending), all the formal CORBA specifications are written in IDL – including even the interfaces of the ORB and its several Object Adaptors (which will be explained later).

Based on independent interface definition of its modules a CORBA application is

- **distributed**, insofar as every module may reside within its own process, eventually on a remote computer or even located within another network
- **heterogeneous**, insofar as every module may be implemented in a different programming language, running within a different operating system, and different hardware
- **object oriented**, insofar as CORBA IDL defines full flavoured Objects to be called using its operations (CORBA jargon for methods)

- **modular**, insofar as every interface implemented by a specific module can (and should) be designed to be as simple and atomic as possible, so that a flexible infrastructure even for highly volatile business processes can be provided.

The CORBA specification is embedded in another embracing architecture, the **Object Management Architecture (OMA)**:



The basic components of OMA are:

- **Common Object Services (COS)** contain standardized interfaces for building a sound technical infrastructure. These services span from the simple hierarchical *Naming Service* (which should be delivered as basic service with every ORB implementation) over useful tools like the *Event & Notification Service* (a distributed version of the well known Observer pattern¹) up to advanced services like *Transaction Service* (a distributed 2-phase-commit) and *Security Service* (embracing all aspects of secure distributed computing).
- **Common Facilities** contain so-called “horizontal” component interfaces for building a business infrastructure, which are useful within every specific business domain. Examples are the *Printing Facility* (standardized access to distributed print services),

¹ s.f. Gamma, Helm, Johnson, Vlissides: *Design Patterns*. Addison Wesley, 1995

Internationalization & Time Facilities, the Mobile Agent Facilities, and The Meta Object Facility (for building all kinds of repositories based in IDL).

- **Domain Facilities** contain so-called “vertical“ component interfaces which are useful (if not necessary) in specific business domains. Examples are the *General Ledger, Currency* and *Party Management Facilities* from *CORBAfinance*, or the *Audio/Video Streams Facilities* from *CORBAtelecom*.
- **Application Objects** are the application-specific component interfaces (i.e. our business as software developers).

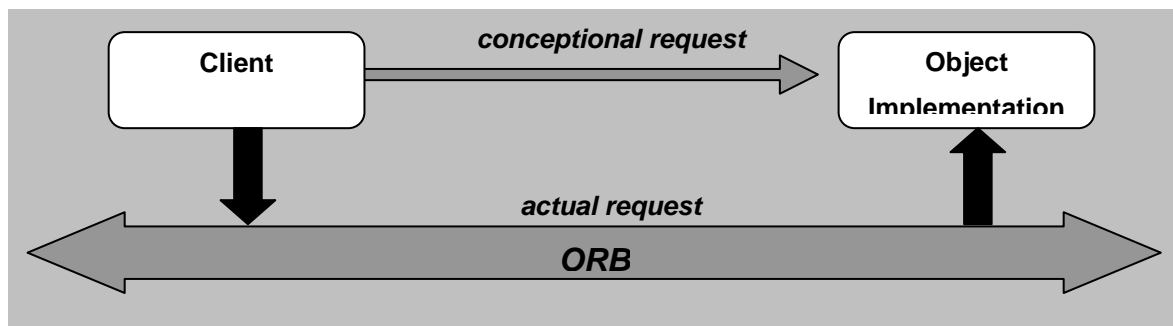
and last but not least

- the **ORB** as the communication medium between all these components.

Common Object Services as well as the several Facilities (and of course, the application objects) are, like the ORB itself, defined in terms of IDL. This means that most of the services & facilities are implemented as CORBA objects (a rare exception is, e.g. the Security Service, as far as there are internal features of the ORB implementation specified). Because of this, the services should be (like all CORBA objects) compatible with all ORB implementations.

Inside CORBA

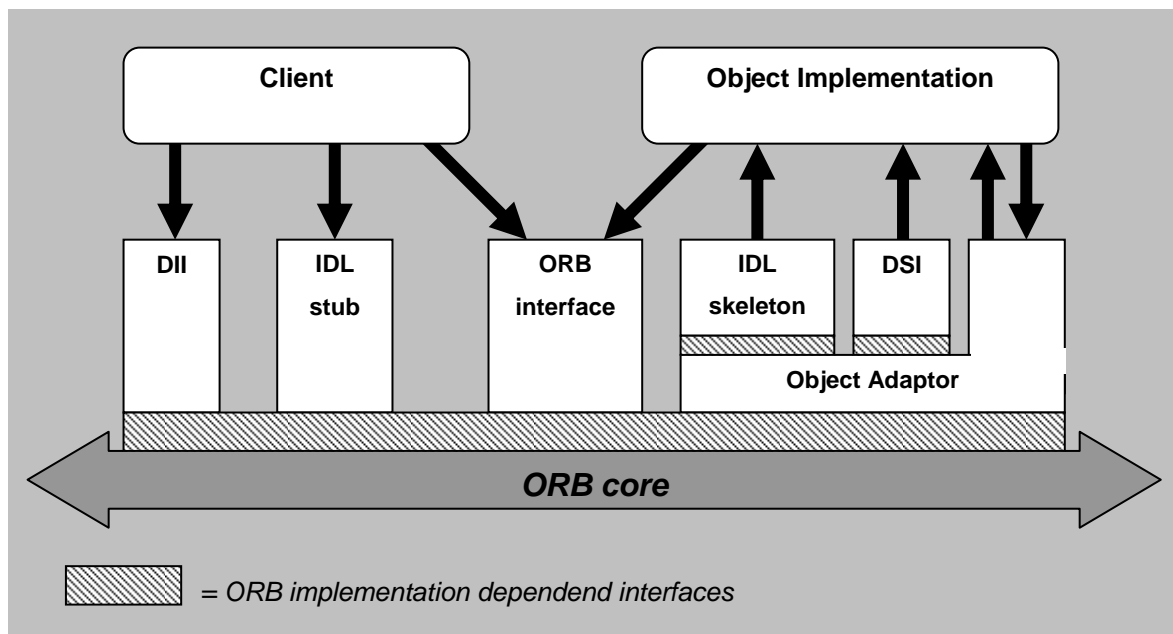
The basic idea behind CORBA (and likewise most other Component Architectures) is the concept of locational transparency: It should make no difference whether an object is called within the same process or on a remote process:



To realize this feature, CORBA uses (as do most other Component Architectures) the Proxy pattern². A CORBA implementation will use the components IDL description to generate two proxies:

- the **Stub**, which encapsulates all the communication codes on the client side. The client will locally instantiate the stub instead of the real object
- and the **Skeleton**, which provides the same functionality on the server side. The object implementation will usually be within a class derived from the skeleton, or will be attached (“tie style”) as a delegate to another, also a generated object, implementation (which in turn is derived from the skeleton).

Of course, before an object implementation can be called, the client (as well as the server) have to connect to the communication infrastructure – the ORB. Therefore, both have to make use of the standardized ORB interface provided by the ORB implementation:



There is some functionality not used by a client (e.g. special policies with regard to the creation of object references or activation modes), in particular one generic interface, the **Object Adaptor**, which is only used on the server side.

² s.f. Gamma, Helm, Johnson, Vlissides: *Design Patterns*. Addison Wesley, 1995

At present, most ORB implementations use the **Portable Object Adaptor** (POA) which is the only mandatory Object Adaptor. (Older applications may depend on the former Basic Object Adaptor which has meanwhile been deprecated.) In the future, additional Object Adaptors may be specified.

The scenario for establishing communication between client and object implementation is pretty straight forward:

- The server process instantiates an object implementation and attaches it to the Object Adaptor.
- The Object Adaptor generates a globally unique ID for this object, an **Interoperable Object Reference** (IOR). The IOR encodes
 - the physical address of the server process,
 - the relative address of the object implementation within the address room of the server process,
 - the interface type & version.
- The server process publishes the IOR (maybe using the *Naming Service*, maybe using a flat file, or any other appropriate way.)
- The client retrieves the IOR and creates a new object from it using operations from the ORB interface. Up to this moment, the new object is of type 'CORBA::Object'. In order to use the proper interface the object has to be downcasted ("narrowed") using some generated helper operations. At this time, the stub is instantiated and returned. During this procedure, a handshake protocol between client stub and object implementation ensures a *typesafe* downcast.
- the client calls the object implementation via its local stub.

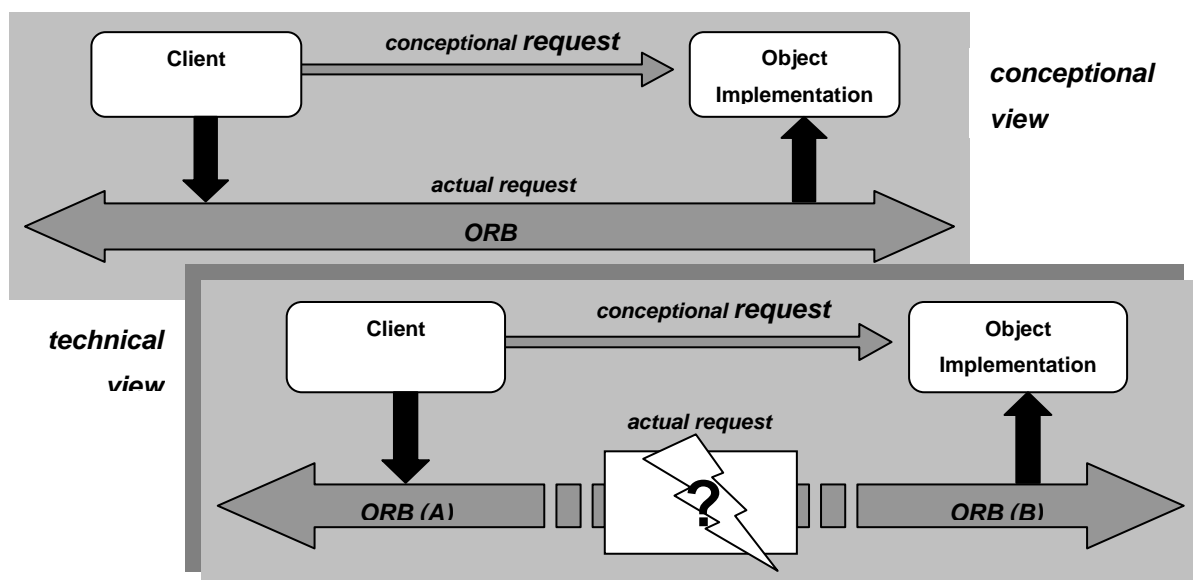
DII and **DSI** are alternatives to the statically generated stubs and skeletons:

Before a DII call can be executed, the IDL description of a specific interface has to be loaded into an *Interface Repository* (another CORBA Object Service). A client can then retrieve the object's interface type (encoded within the IOR) using the generic *CORBA::Object* interface and access all necessary meta information to compose a generic request using the CORBA **Dynamic Invocation Interface** (DII) to the object implementation – without a generated static stub.

The analogous server's side mechanism is defined by the **Dynamic Skeleton Interface (DSI)**. It provides a way for delivering requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. This is primarily of use within dynamically typed languages like CommonLisp or Smalltalk.

CORBA interoperability (IIOP)

The interoperability and flexibility of CORBA makes it a perfect integration solution bridging the gaps between different programming languages, operating systems, computer hardware. The assumption, of course, is that the interoperability of different ORB *implementations* is guaranteed:



Every CORBA implementation uses the standardized **General Inter-ORB Protocol (GIOP)** for communication with other ORBs. The GIOP specifies the marshalling (serializing) and unmarshalling of CORBA requests and replies. The most common specialization of the GIOP is the **Internet Inter-ORB Protocol (IIOP)** for CORBA communication over TCP/IP.

3 Welcome to MinCor.NET

MinCor.NET is a CORBA Object Request Broker written in C#. It runs as managed code on the Microsoft .Net platform and complies with the specification published by the Object Management Group.

The C#-language mapping (its formal OMG specification is still pending) is kept similar to the Java-language mapping. The main reason for this approach are conceptual similarities between C# and Java. Both languages support interfaces with multiple inheritances which require explicit implementation by classes. A Java or C# class, on the other hand, can only inherit from a single base class. It therefore seems obvious to identify CORBA interfaces with C# interfaces. As in Java all administrative functions associated with an IDL interface are made available as part of an additional generated class with the suffix 'Helper'.

The major difference between the Java- and the C#-language mapping is the absence of 'Holder' classes. C# allows for in/out-parameters and call-by-reference as compared to Java. These C# features make the introduction of extra container or holder classes redundant.

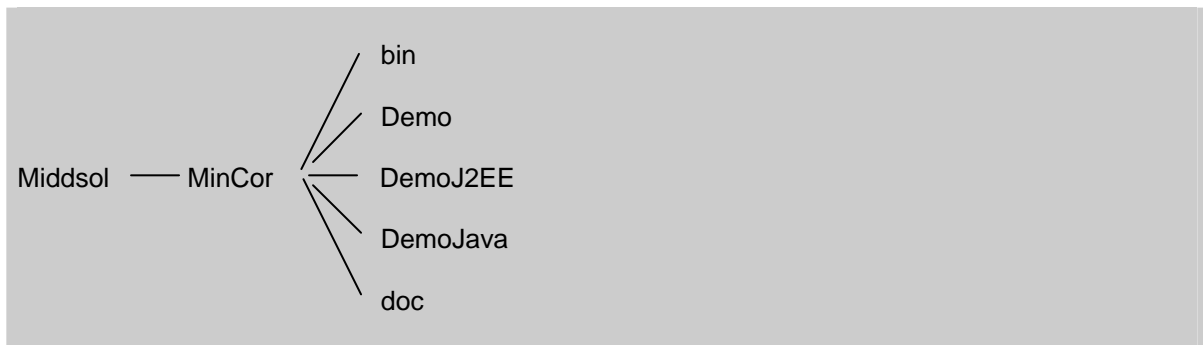
CORBA types are directly mapped to C# types as far as possible. More details on the type mapping is given in the following sections.

The language independent concept of the .Net-framework allows the use of several languages within one application. Although MinCor.NET has been implemented in C# and the corresponding IDL compiler produces C# code the application itself can be written in C++ or Visual Basic (or other .Net-based languages). The generated C# code can be compiled into a separate dynamic link library which can be linked with any C++ or VB application.

The overall guideline for using C# may easily be deduced from Java based CORBA applications. A comprehensive discussion on the Java language mapping can be found in the literature. (A good introduction may be found in *Brose, Vogel, Duddy: Java Programming with CORBA. Wiley & Sons 2001*. For a more straight- forward approach, have a look at the originall OMG *IDL to Java Language Mapping Specification*, which can be found for download at <http://www.corba.org>.)

4 Installing MinCor.NET

Having executed the MinCor.msi installation file you find the following folders in the MinCor.NET home directory:



Please, add the full path `..\Middsol\MinCor\bin` to your PATH-environment variable.

The bin-folder holds the idl-compiler **MCidl2cs.exe** and the dynamic link library **MinCorCF.dll** which contains the ORB. Once you start a new project you have to include the MinCorCF.dll in your **reference** list in order to successfully build your application.

When you move the MinCorCF.dll to a new location you have to reset the reference in your project.

5 Open Issues

MinCor.NET version 2.0 is one release in a series of C# ORB releases. It contains all major functionalities required in a CORBA environment, but does not support the full range of the OMG specifications yet.

Not supported IDL types are 'fixed types', 'native types' and 'Dynamic Any'. All other types including 'Any' and 'value types' are implemented.

Missing features will be added in future releases.

6 A Simple CORBA Application

We will demonstrate the use of MinCor.NET with the help of a simple example. We create two .Net compact applications, a server and a client. The server provides a certain functionality which the client will call using CORBA.

The examples given assume the use of the Windows Mobile platform (which does not support console applications). MinCor.NET, however, is not restricted to the Windows Mobile platform and runs on any Windows CE platform.

Step 1. The IDL File

Any CORBA based design begins with the layout of the interfaces needed by the application. All interfaces and additional types are defined in an IDL file. In our 'hello' example we have one interface 'Greetings' with a single method 'hello'. The parameter is a string:

```
// hello.idl
module Example
{
    interface Greetings
    {
        void hello( in string strName );
    };
};
```

The IDL source is stored in the file 'hello.idl'. In order to generate code one has to call the IDL compiler MCidl2cs :

```
> MCidl2cs.exe hello.idl
```

The IDL compiler creates a file 'hello.cs' containing stubs, skeletons, and several helper classes.

Step 2. The Server Code

It takes only a few steps to create the server part of the application:

1. Create a new C# **mobile application** for the server.
2. Add the generated source file **'hello.cs'**.
3. Add a reference to the **'MinCorCF.dll'**, the MinCor.NET ORB runtime located in the MinCor.NET bin directory.
4. Implement the server code.

One of the central tasks for writing the server code is the implementation of the interface 'Greetings'. The generated class 'GreetingsPOA' which serves as base class for 'GreetingsImpl' provides a link between the object implementation and the object adapter.

Within the constructor of the 'ServerImpl' we

- initialize the ORB and the Portable Object Adapter (POA),
- instantiate the 'GreetingsImpl' and register the servant,
- export the Interoperable Object Reference (IOR)
- and call the ORBs run() method, which starts the main event loop waiting for external CORBA requests.

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;

namespace HelloSrv
{
    public class ServerImpl
    {
        private IFrm                m_oFrm;
        private Midsol.CORBA.ORB    m_oOrb    = null;
        private string[]            m_strORBInit =
            { "-ORBEndpoint iiop://localhost:8545/portspan=100",
              "-ORBDebug Out=.\\HelloSrv.log" };

        public ServerImpl( FrmHelloSrv a_oFrm)
        {
            m_oFrm = a_oFrm;

            // initialize the Orb, listening on the TCP/IP port 8545
            // (or one of the following 100 port numbers, if 8545 is not available...):
            m_oOrb = Midsol.CORBA._ORB.init( m_strORBInit, null );
        }
    }
}
```

```
// retrieve the POA from the Orb:
Middsol.PortableServer.POA oRootPOA
    = Middsol.PortableServer.POAHelper.narrow(
        m_oOrb.resolve_initial_references( "RootPOA" ) );

oRootPOA.the_POAManager.activate();

// create & register a GreetingsImpl:
Middsol.CORBA.Object oObjRef
    = oRootPOA.servant_to_reference( new GreetingsImpl( this ) );

// publish the IOR:
Middsol.CORBA._ORB.wrlORtoFile( ".\HelloSrv.ior", oObjRef);

m_oFrm.writeLog( "Server running");
}

public void destroy()
{
    if( m_oOrb != null)
    {
        m_oOrb.destroy();
        m_oOrb = null;
    }
}

public IFrm theFrm
{
    get{ return m_oFrm; }
}
}

// The Implementation:
public class GreetingsImpl : Example.GreetingsPOA
{
    private ServerImpl m_oServer;

    public GreetingsImpl( ServerImpl a_oServer)
    {
        m_oServer = a_oServer;
    }

    override public string hello( string a_strName )
    {
        m_oServer.theFrm.writeLog("Method executed:");
        m_oServer.theFrm.writeLog(" GreetingsImpl:hello");
        m_oServer.theFrm.writeLog(" Parameter:" + a_strName);
        return "Hallo " + a_strName ;
    }
}
}
```

Note, that within a Windows Mobile application, only the main thread may call the GUI controls. Therefore, all other threads have to use a queue to communicate with the main thread, which in turn accesses the GUI for output. Here, this is done within the frames 'writeLog()' method:

```
public void writeLog( string a_strMsg)
{
    System.Threading.Monitor.Enter( m_oSync);
    m_oMsgQueue.Enqueue( a_strMsg);
    System.Threading.Monitor.Exit( m_oSync);
}
```

Step 3. The Client Code

The client is created in a similar way:

1. Create a new C# **mobile application** for the client.
2. Add the generated source file '**hello.cs**', which contains CORBA stubs and skeletons.
3. Add a reference to the '**MinCorCF.dll**', the MinCor.NET ORB runtime located in the MinCor.NET bin directory.
4. Implement the client code.

The client reads the IOR from a file which the server created at start-up. With the help of the IOR the client can connect to the server and call the function 'hello'.

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;

namespace HelloClt
{
    public class ClientImpl
    {
        private Example.Greetings    m_oGreetings = null;
        private Midsol.CORBA.ORB      m_oOrb       = null;
        private string[]              m_strORBInit =
            { "-ORBEndpoint iiop://localhost:8645/portspan=100",
              "-ORBDebug Out=.\\HelloClt.log",
              "-CltSndTimeout 1000"};
    }
}
```

```
public ClientImpl()
{
    // initialize the Orb:
    m_oOrb = Midsol.CORBA._ORB.init( m_strORBInit, null );

    // retrieve the CORBA object using the published IOR file:
    m_oGreetings = Example.GreetingsHelper.narrow(
        m_oOrb.string_to_object("file://.\HelloSrv.ior" ) );
}

public void destroy()
{
    if( m_oOrb == null) return;
    m_oOrb.destroy();
}

public Example.Greetings Greetings
{
    get{ return m_oGreetings;}
}
}
}
```

To start the communication between client and server, we retrieve the 'Greeting' reference from the 'ClientImpl' and call its 'hello()' method:

```
private void btSend_Click(object sender, System.EventArgs e)
{
    if( m_oClientImpl != null)
    {
        // fill the TextBox' Text attribute:
        txtReturn.Text = m_oClientImpl.Greetings.hello( txtTextLine.Text);
    }
}
```

Step 4. Running the Application

Copy both applications, client and server, into a common directory of the target device. After starting the server and the client, the screens look as follows:



7 IDL-to-C# -Mapping

In this section we will give a brief overview of the MinCor.NET IDL-to-C# language mapping.

Most IDL types have a direct C# counterpart. IDL identifiers will be directly mapped into C# identifiers and a module is represented by a C# name space. C# keywords which are used as IDL identifiers are prefixed by an underscore. Interfaces, Structures and Unions receive an additional name space with the suffix 'Package' to accommodate new definitions of types within the corresponding type or interface name spaces.

Basic Data Types

The following table gives a listing of the basic IDL data types and their representation in C#:

IDL Type	C#
boolean	bool
char	char
wchar	char
string	string
wstring	string
octet	byte
short	short
unsigned short	ushort
long	Int
unsigned long	uint
long long	long
unsigned long long	ulong
float	float
double	double
long double	double

Interface

Interfaces are directly mapped to their C# counterparts. 'Attributes' defined in the IDL interface become C# properties.

Helper Classes

Similar to the Java mapping, we generate .Net Helper Classes for any user defined type (using the type name with the suffix 'Helper'). Helpers will provide a couple of static methods

- inserting and extracting user type objects into and from CORBA::Any objects,
- writing user type objects into and reading them from the stream,
- 'narrowing' user type objects from CORBA::Object into their proper type.

Most of the time, only the last functionality is needed by the developer.

Typedef

Similar to Java, the IDL statement 'typedef' does not introduce a new type in C#. It forces only 'Helper' classes to be generated. Since C# does not know 'typedefs', the original type has to be used in the application code. The 'Helper' class provides the repository id for the corresponding CORBA type.

Constant

C# does not allow for a global definition of constants. CORBA constants are therefore represented as 'structs' with one member called 'value', i.e. the constant is set by assigning `<constant>.value = <some value>`.

Constructed Data Types

Constructed types comprise structures, enumerations and unions.

- **Struct:** IDL structures are mapped to native C# structures. Members in C# structures can be initialized via a constructor with the members as parameters. A standard constructor which is always present in C# can also be used. In this case the members have to be initialized individually.
- **Enum:** Enumerations have a C# equivalent.
- **Union:** Unions are mapped to C# classes. Each union branch is associated with a C# property which allows a direct assignment of values to each branch. This way the use of unions is very much like that of structures.

Array and Sequence

Arrays as well as sequences are mapped to C# arrays. They usually appear in conjunction with 'typedefs' or 'constructed types'.

CORBA Pseudo Objects

A CORBA Object is by definition a remote object offering a remote interface to the client.. Nevertheless, there are some objects defined in terms of IDL (for the sake of correctness: *PIDL – Pseudo IDL*) within the CORBA standard, which are *not* remote but necessarily local, like the ORB interface, or objects which are mandatorily transported by value:

- The **ORB** is mapped into **Middsol.CORBA.ORB**
- The **POA** is mapped into **Middsol.PortableServer.POA**
- The type **Any** is mapped into class **Middsol.CORBA.Any**
(Any may be viewed as the IDL pendant to the 'void' pointer.)
- **CORBA::Object** (as superclass for all CORBA stubs as well as a locally used class) is mapped into **Middsol.CORBA.Object**

8 Tools

The IDL Compiler - MCidl2cs.exe

The IDL compiler is the most important tool (besides the runtime libraries) for any CORBA ORB implementation. It transforms the IDL description into the target language.

The MinCor.NET IDL compiler target language is C# using the mapping described above (s.f. page 17). The generated code can easily be compiled into a separate DLL which is then linked into the .Net application in question. This way MinCor.NET can be used with all .Net capable languages.

Usage: **MCidl2cs.exe [flag ...] [file ...]**

Legal options are:

- **-all** Emit code for the included idl-files as well.
- **-d <dir>** Set the destination path to <dir>.
- **-h** Print help message.
- **-I <dir>** Include <dir> in the search path for the preprocessor.
- **-keep** Do not overwrite existing file.
- **-noBuiltInTypes** Built in types are not included.
- **-noDefaultFactories** No default factories for value types are generated.

- **-noDefImpl** No default implementations for value types are generated.
- **-noPOA** No POA skeleton classes are generated, only client side stubs.
- **-noTIE** No TIE style object implementation classes are generated.
- **-prefix <name> <pref>** Extend the name <name> of a module at root level by the prefix <pref>.
- **-v** Trace compilation stages.
- **-V** Print version info.
- **-w** Suppress IDL compiler warning messages.
- **-Wp,<arg1>,<argn>** Pass these arguments to the preprocessor.

URL formats

CORBA objects may be addressed using various URL formats. The URL address is interpreted immediately by the ORB using its `string_to_object()` method.

Currently MinCor.NET supports three different URL formats:

- **IOR:<hex octets>**,
the hexadecimal encoded IOR address, e.g.: "IOR:011457002100000049444c3e6f726..."
- **file://<path>**,
referring to a file containing the IOR, e.g.: "file://c:\\hello.ior" or "file://c:/hello.ior"
- **corbaloc: [iiop | ssliop] : [<version>] <host>: <port>/<key_string>**,
with
 - **version** = optional version number "<major>.<minor>@" (default: "1.0@")
 - **host** = host address (DNS name or IP address, default: "localhost")
 - **port** = TCP/IP port of the server object
 - **key_string** = name of the server objecte.g.: "corbaloc:iiop:1.2@armageddon:1050/NameService"

9 Runtime Configuration

The ORB can be configured by an array of parameters passed through the `init()` function. Some of the parameters are OMG standards.

Standard parameters:

- **-ORBid** <orb id>, identifier for an ORB instance.
- **-ORBInitRef** <ObjectID>=<ObjectURL>, sets the object URL for an initial object reference..
The most prominent example is the naming service:
 - `-ORBInitRef NameService=corbaloc:iiop:1.1@localhost:1600/NameService`
 - `-ORBInitRef NameService=file://c:\ns.ior`
 - `-ORBInitRef NameService=IOR:00001....`
- **-ORBDefaultInitRef** <ObjectURL>, the ObjectURL is appended by a '/' and the object key of the service given in **resolve_initial_references**.

Non-standard parameters:

- **-ORBEndpoint** `iiop://<hostname>:<port>[/portspan:<x>]`, sets the hostname and port for the server socket. If the <port> is not available the server searches for another port in the range given by the 'portspan'.
- **-ORBDebug** [**Out**=<file name> (or) **Append**=<file name>] [**Level**=<level>], enables debugging on different levels. When no file name is given, a log file is created with a name that incorporates the task name and process id. Using the parameter **Out** will result in overwriting an existing file whereas **Append** adds the debug output to an existing file. The following levels can be applied: **Level=1** (critical exceptions), **Level=2** (tracing of connections), **Level=5** (tracing of requests/replies), **Level=6** (GIOP message dumps), **Level=10** (Fragment dumps), **Level=11** (SSL/TLS message dumps).
- **-ORBKeyAlias** <alias>=**RootPOA**[/<POA chain>]/-**MC**-/<object key>, defines on the server side an alias for the object key. <POA chain> represents the addressed POA as optionally created by the RootPOA and its possible descendants. This right side of this parameter is subject to possible future changes. Other means for setting a key alias can be found in the document 'MinCor.NET.chm', namespace 'CORBA.Object'.

- **-AddAssembly=<comma separated assembly list>**, adding one or more assemblies to be considered by the .Net Reflection (used within MinCor.NET in the context of handling CORBA valuetypes.)
- **-TP=<n>**, number of threads.
- **-InBuf=<size>**, minimum buffer size in bytes for incoming messages.
- **-OutBuf=<size>**, minimum buffer size in bytes for outgoing messages.
- **-IIOPVers=<n.m>**, defines the GIOP Version the ORB is running with.
- **-BigEndi=<true/false>**, defines the Endianess of the ORB at runtime.
- **-ConType=<TLS/TCP >**, enables/disables the secure socket layer (default=TCP).
- **-P12Filename=<filename>**, password protected PKS#12 file containing a certificate and a key for server authentication.
- **-P12Password=<password>**, password for the PKS#12 file.
- **-LocReq=<true/false>**, defines whether the ORB issues a locate request for any new object.
- **-ClitConnection=<PerClient,PerObject>**, one connection per client or one connection per object, the default is one connection per client.
- **-ClitRecTimeout=<timeout in mecs>**, sets the client side timeout for receiving a reply.
- **-SuppressCodeset=<true/false>**, controls the codeset handling, the default is false. When set to 'true' the Windows default setting is assumed and no codeset information is send in the service context.
- **-SrvIdleTimeout=< timeout in msec >**, sets the idle timeout of a server. After the idle time interval has elapsed the server sends a 'close connection' message to the client.
- **-ConnectRetries=<number of attempts>[:<time interval between attempts in msec>]**, sets the number of connection attempts. In the second part of the parameter, separated by a colon, a time interval between the attempts can be inserted. The default for the time interval is 500 msec.

10 Client Connection Model

MiddCor offers the possibility to choose between two models for a client connection (controlled by the parameter **-CltConnection=<PerClient,PerObject>**):

1. one connection per client/server pair (default option)
2. one connection per object

In the first case the connection to a server is running over one channel for all objects, in the second case each object creates its own channel, i.e. with each new object a client socket is opened. The first option which is the default option requires less resources and is the best choice for most environments. Depending on the threading model of the server, however, and to mention is in particular a threading model with only one thread per connecting client, the performance is heavily increased when each object is using its own channel (connection). In this scenario one should use the option 'one connection per object'.

MiddCor's server implementation provides a thread pool (The number of threads is controlled by the parameter **-TP=<n>**). With this threading model each request is served by a separate thread independent of the client connection. The performance is only limited by the number of threads in the pool.

11 Transport Layer Security (TLS)

Securing data communication is one of the essential corner stones for mobile computing. MiddCor.NET allows to secure communication sockets using TLS. The TLS handshake protocol and the encryption is handled by a security service based on technology provided by [Mono](#). For TLS to work certificates are required. Trusted certificates need to be installed in the **'/Middsol/ApplicationData/mono/certs/Trust'** folder. When MiddCor.NET act as client and tries to establish a TLS connection any received certificate is checked against the trusted certificates located in the 'Trust' folder. These certificates need to be binary coded (X.509) and stored in files with the extension **.cer**. When a secure server socket is to be opened, i.e. the parameter **-ConType=TLS** is set, MiddCor.NET itself requires a certificate to be presented to a potential client. This certificate (including an optional certificate chain) and the accompanying private key need to be stored as a password protected PKCS#12 file. The flags **-P12Filename=<filename>** and **-P12Password=<password>** allow to access the certificate.

Client authentication is not supported with the current Mono version (1.0.5) and it is recommend to use TLS instead of SSL. TLS is the follow-up protocol for SSL.

Certificates can be exported from a Windows system using the tool 'certmgr'.

12 User controlled Debug output

The method `_ORB.init` allows to pass a third parameter that contains a reference to a user defined text stream. This parameter needs to be an implementation of the class `System.IO.TextWriter`. The method `WriteLine` of this implementation is used to output debug statements. Overwriting this method allows to control the output.

The method `setDebugLevel` located in the ORB-interface can be used to change the debug level at runtime.

13 Sources and References

Web References

- The OMG homepage: <http://www.omg.org/>
- The OMG CORBA homepage: <http://www.corba.org/>
- Douglas Schmidt's CORBA page (may be reached via Doug Schmidts personal homepage): <http://www.cs.wustl.edu/~schmidt>

Books

- Brose, G., Vogel, A., and Duddy, K.: *Java Programming with CORBA*. Wiley & Sons, 2001.
- Gamma, E, Helm, R, Johnson, R., and Vlissides, J.: *Design Patterns*. Addison-Wesley, 1995.
- Mowbray, T., and Malveau, R.: *CORBA Design Patterns*.Wiley, 1996.
- Orfali, R, Harkey, D., and Edwards, J.: *Instant CORBA*. Wiley, 1997.

- Ruh, W., Herron, T., and Klinker, P.: *IIOP Complete. Understanding CORBA and Middleware Interoperability*. Addison-Wesley, 1999.
- Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F.: *Pattern-Oriented Software Architecture 2. Patterns for Concurrent and Networked Objects*. Wiley, 2000.
- Siegel, J.: *CORBA – Fundamentals and Programming*. Wiley, 1996.
- Slama, D., Garbis, J., and Russell, P.: *Enterprise CORBA*. Prentice Hall, 1999.