

IIOP.NET architecture documentation

Dominic Ullmann

September 19, 2004

Abstract

This document presents the architecture of IIOP.NET.

Contents

1	Architecture	3
1.1	Overview	3
1.2	IIOPChannel architecture	4
1.2.1	.NET remoting channel implementation layer	4
1.2.2	Giop message layer	5
1.2.3	Type marshalling layer	5
1.2.4	Cdr representation layer	5
1.3	IDL to CLS compiler architecture	6
1.4	CLS to IDL generator architecture	6
1.5	extension points	7
1.5.1	Mapping	7
1.5.2	Transport protocols	7

List of Figures

1.1	IIOP.NET main parts	3
1.2	IiopChannel: Layers	4
1.3	IDL To CLS compiler main parts	6
1.4	IDL To CLS compiler: processing files	8

Chapter 1

Architecture

1.1 Overview

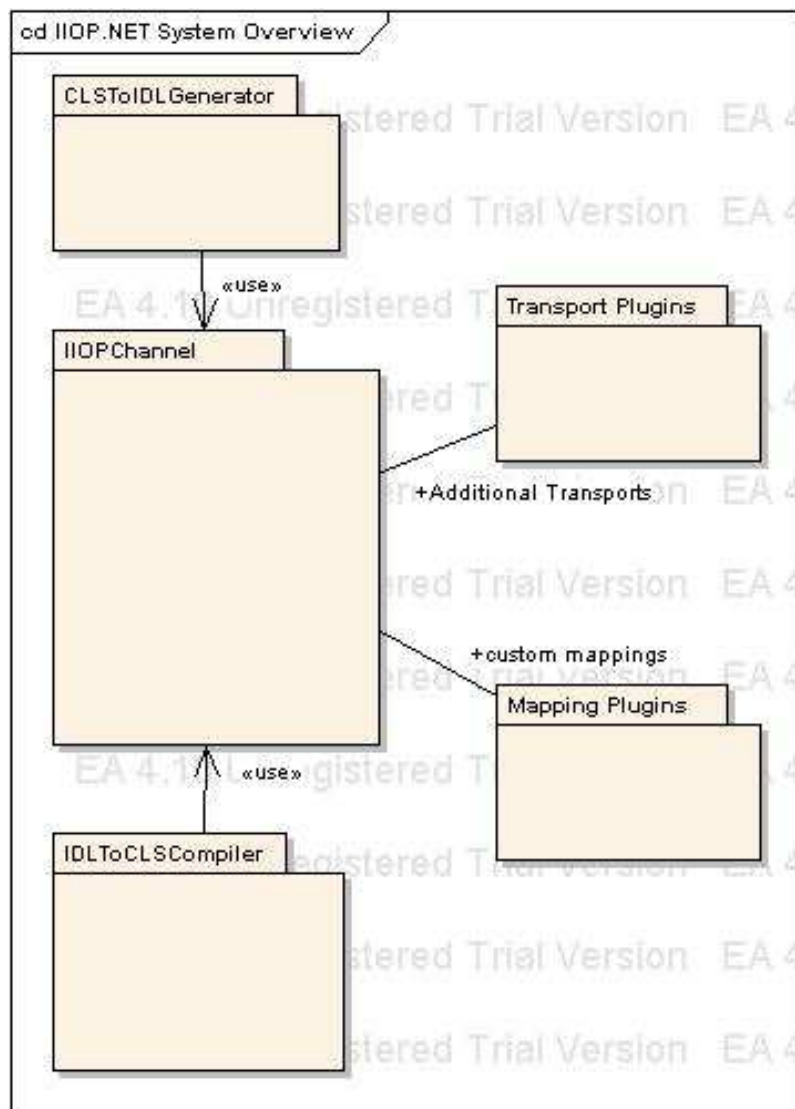


Figure 1.1: IIOP.NET main parts

IIOP.NET consists of the following 3 main parts:

- IIOPChannel: The .NET remoting channel, allowing to use CORBA/IIOP as remoting protocol.
- IDLToCLSCompiler: creates .NET assemblies for CORBA IDL.
- CLSToIDLGenerator: creates CORBA IDL for .NET assemblies.

1.2 IIOPChannel architecture

In this chapter the architecture of the IIOPChannel is detailed.

Conceptually, the IIOPChannel consists of the layers detailed in the following figure 1.2.

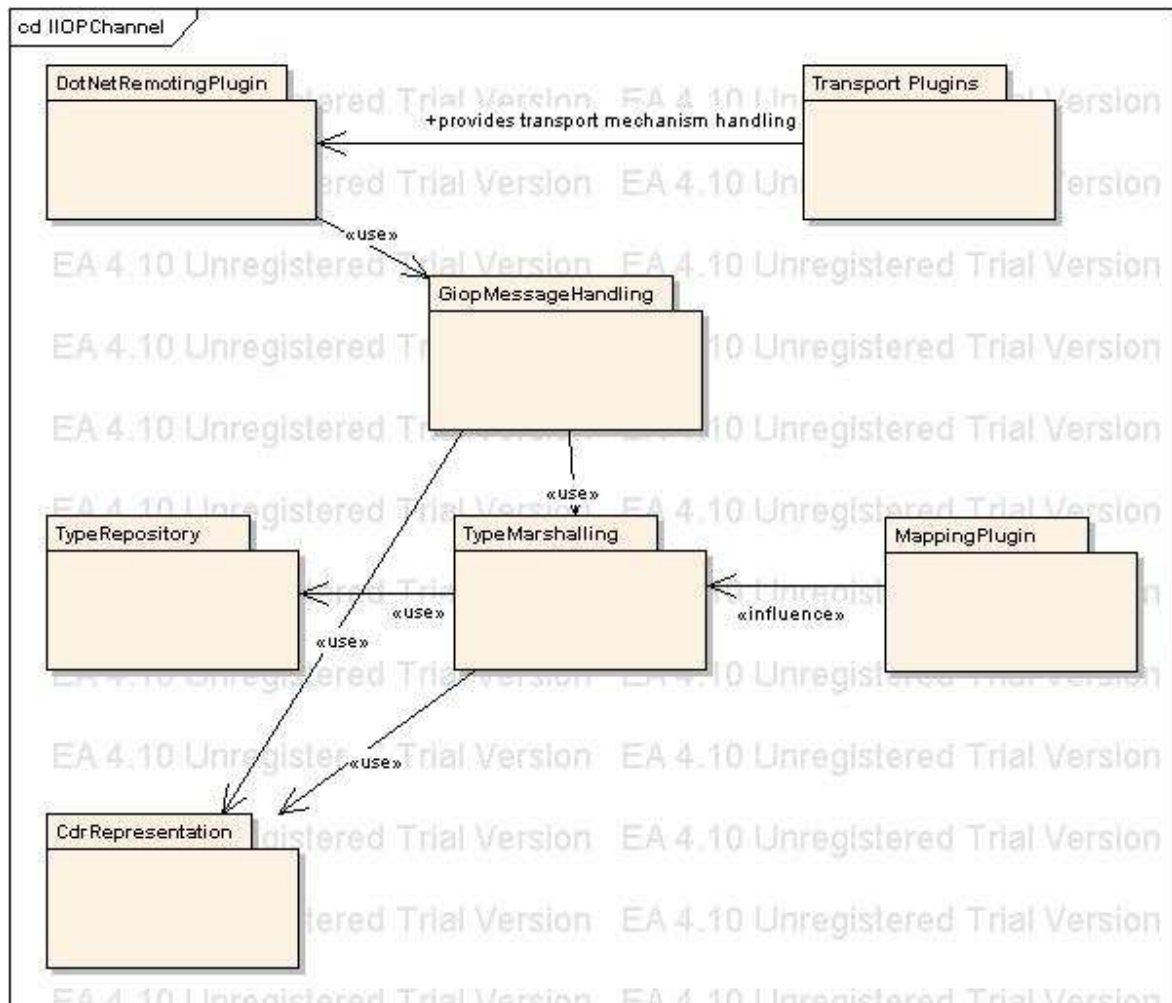


Figure 1.2: IiopChannel: Layers

1.2.1 .NET remoting channel implementation layer

This layer contains the classes, which implement the classes needed for a .NET remoting channel, i.e.

- The channel classes: *IiopChannel*, *IiopServerChannel*, *IiopClientChannel*
- The formatter sink classes: *IiopClientFormatterSink*, *IiopServerFormatterSink* and the formatter sink providers: *IiopClientFormatterSinkProvider*, *IiopServerFormatterSinkProvider*

- The transport sink classes: `IiopClientTransportSink`, `IiopServerTransportSink` and the sink provider for the client side: `IiopClientTransportSinkProvider`

The `IiopServerChannel` is a channel receiver. It listen for incoming request messages and sends reply messages. The `IiopClientChannel` is responsible for sending request messages and receiving the replys. The `IiopChannel` wraps those two channels to provide a channel, which is able to send and receive messages. This wrapper channel allows for example to use callback objects. In this case, the client channel part sends the message containing references to the callback object. The sever channel part is then responsible for listening for requests to the callback object.

The formatter classes are responsible for creating the GIOP-messages from .NET remoting messages and for creating .NET remoting messages from GIOP-messages. The formatter classes use the services provided by the GIOP message layer to accomplish this task.

The transport classes are responsible for sending and receiving GIOP messages with/from a transport mechanism. Because some functionality is common to all transport mechanisms, e.g. the reassembling of fragmented GIOP messages, the transport specific parts are realised with a plugin mechanism. The `IiopChannel` already contains the default plugin for the TCP transport. A transport plugin for SSL is also available.

1.2.2 Giop message layer

This layer is responsible for the handling and the serialization/deserialisation of GIOP Messages. The `GiopMessage` The class `GiopMessageHandler` gets .NET remoting messages to send from the formatter. It prepares the serialisation and delegates this job this job to the `GiopMessageBodySerialiser`. The `GiopMessageHandler` delegates the deserialisation of `GiopMessages` to the `GiopMessageBodySerialiser` and handle GIOP messages without a counter part in .NET remoting, e.g. `ForwardRequest`. It completes the reception job by passing back a .NET remoting message to the formatter.

1.2.3 Type marshalling layer

This layer is responsible for marshalling .NET types as CORBA types and to unmarshal CORBA types to .NET types.

The `Marshaller` class decides on how to marshal/unmarshal a certain type with the help of the `CLStoIDLMapper` class and the `SerializerDetermination` class. It delegates the serialisation/deserialisation of the different CORBA types to the responsible `Serialiser`. The `MarshallerForType` class is a specialisation of the `Marshaller`, which decides only once on the mapping for a specific type and than conserves the mapping decision for later reuse.

The Mapping plugin mechanism is able to influence decision of the `Marshaller/CLStoIDLMapper` to allow user defined mappings between CORBA and .NET types, e.g. .NET `ArrayList` and CORBA type for java `Arraylist`.

The `TypeRepository` part is responsible to find/load types for Corba type identifiers (repository id's).

1.2.4 Cdr representation layer

This layer is responsible for creating/parsing the binary representation of primitive CORBA types. It contains the `CdrStream` classes, which can read/write primitive Corba types from/to a stream. Additionally, there are the `CdrEncapsulationStream` classes, which are responsible for handling `Cdr` encapsulations.

1.3 IDL to CLS compiler architecture

In this chapter the architecture of the IDL To CLS compiler is detailed.

The IDL to CLS compiler is mainly composed of a Console UI, an IDL preprocessor, a Parser and Code generator (see figure 1.3).

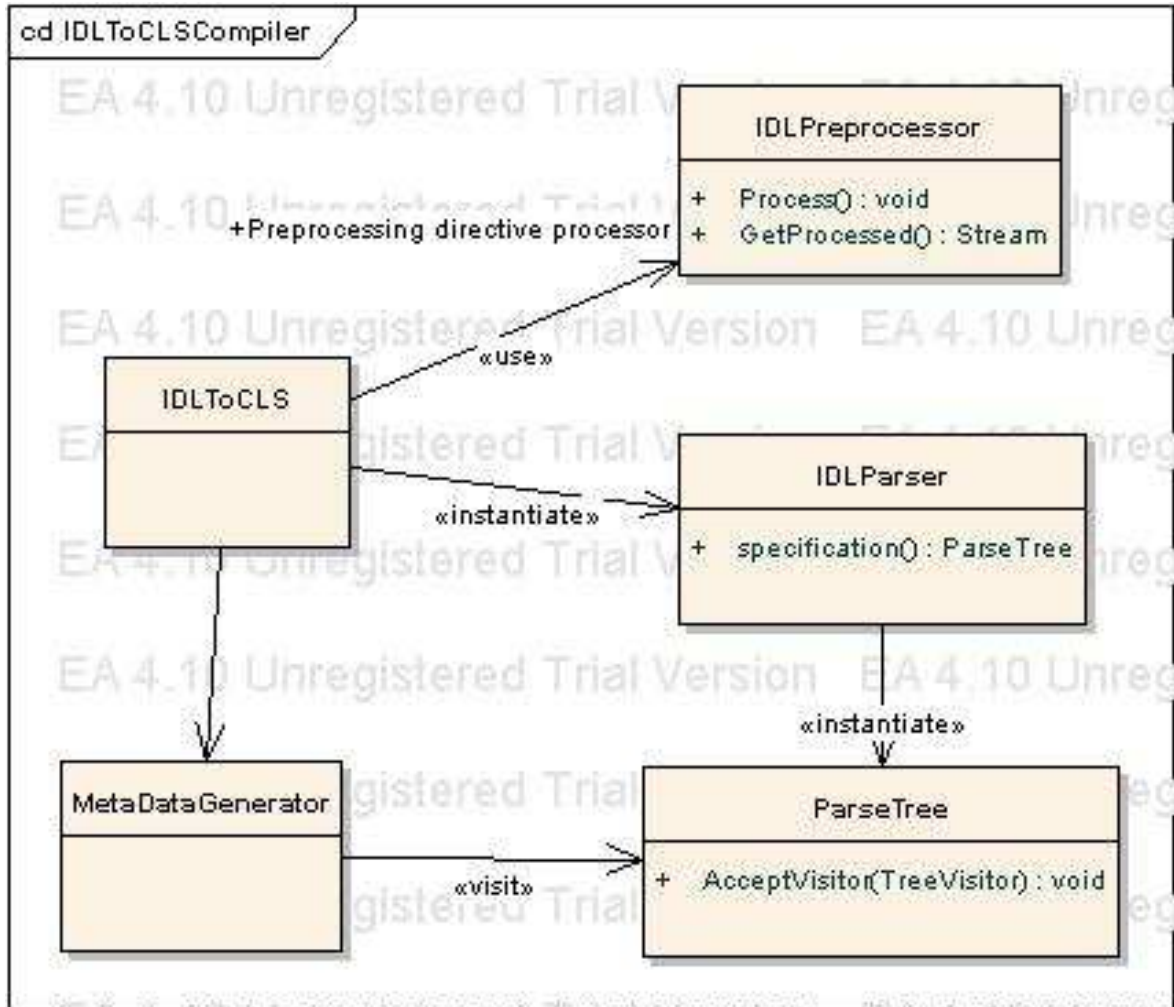


Figure 1.3: IDL To CLS compiler main parts

The class **IDLtoCLS** implements the UI part and delegates the compilation work to the preprocessor, parser and code generator. The **IDLPreprocessor** class handles the IDL preprocessor directives like `include`. It passes a preprocessed stream to the parser for building up an abstract syntax tree (**ParseTree**). This tree consists of nodes representing the different constructs in the IDL. This tree supports the visitor pattern. The **MetadataGenerator** is visitor, which generates .NET CLS for the parse tree. To support building one .NET assembly for multiple independant idl files, this visitor is able to build code for multiple parse trees one after the other into one assembly. The process of building a .NET assembly for multiple idl files is shown in figure 1.4

1.4 CLS to IDL generator architecture

In this chapter the architecture of the CLS to IDL generator is detailed.

1.5 extension points

1.5.1 Mapping

This extension possibility allows to specify custom mappings between .NET and CORBA types. This is useful, if there are similar types in the .NET and in the peer system, but the mapping doesn't map them to each other by default, e.g. .NET ArrayList and java ArrayList.

1.5.2 Transport protocols

This extensions allows to send/receive Giop Messages over/with a specific transport protocol, e.g. TCP or SSL. By default the IIOPChannel contains the Tcp transport protocol plugin. Additionally an SSL plugin is also part of IIOP.NET.

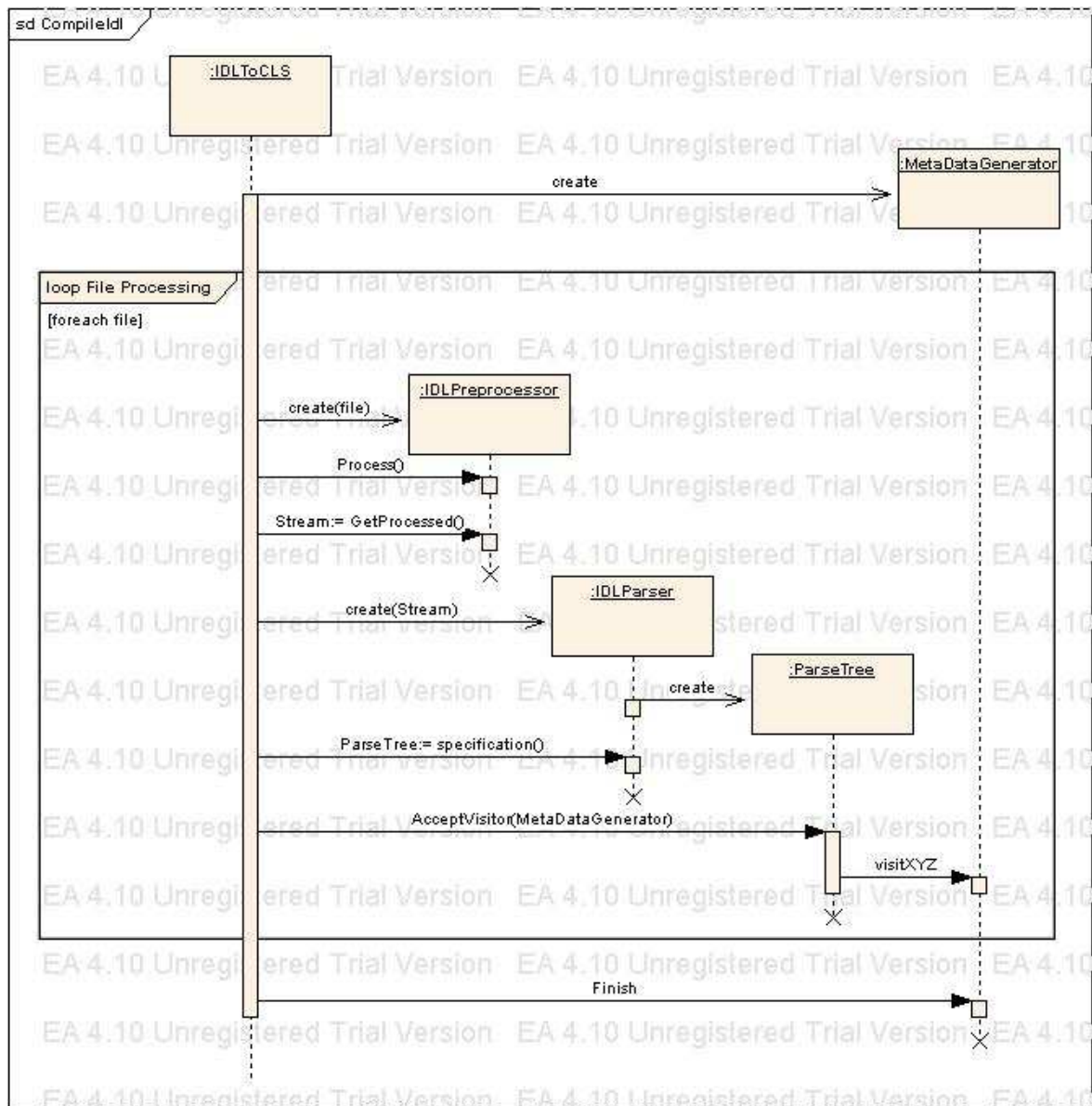


Figure 1.4: IDL To CLS compiler: processing files