Remedy IT

Your challenge - our solution

CORBA Programmers Guide

Remedy IT Expertise BV

Table of Contents

Preface

This is the CORBA Programmers Guide published by Remedy IT. This guide is published for free.

Based on the user feedback and requests we will extend this guide on a regular basis.

Contact information

Remedy IT
The Netherlands

For inquiries related to our services and training you can contact us at sales@remedy.nl or check our website.

When you have a support contract with Remedy IT you can report issues through the Remedy IT Support Portal.

If you have remarks regarding this guide you can send your remarks to corbapg@remedy.nl.

The Remedy IT website is online at https://www.remedy.nl

Chapter 1. Remedy IT Support

For open source products Remedy IT provides flexible Open Source Support options. We provide RS contracts for AXCIOMA, TAOX11, ACE, TAO, R2CORBA, RIDL, and OpenDDS.

With our remote support contracts we can be of assistance for solving issues and answering questions. We provide a support contract which includes email, phone, and web-based access.

We have four different types of remote support contracts so you can choose the kind of support that matches best your support needs:

- RS_Day
- RS_Year
- RS_Month
- RS_SLA

All RS contracts include access to the Remedy IT Software Support Portal. RS is intended for the following type of activities:

- Bug fixing / problem resolution not requiring major redesign
- Small (relatively) functional enhancements and extensions
- Development support (i.e. supplying building, coding, optimization information, example or prototype implementations)

For all these RS contracts it is necessary to sign a support contract which you can request by email from sales@remedy.nl.

For more information about our support see our website.

Chapter 2. Training

We provide several courses as part of our training program. These courses are organized as open enrollment in The Netherlands but also as onsite training at a location of your choice. Check our website for the current course schedules. At this moment we provide the following courses:

- Using the ACE C++ Framework
- Introduction to CORBA
- CORBA Programming with C++
- CORBA Programming with C++11
- Advanced CORBA Programming with TAO
- Component Based Development using AXCIOMA

If you are interested in any of these courses contact sales@remedy.nl.

2.1. Using the ACE C++ Framework

Our ACE course will learn you the concepts of ACE. Through lectures and a set of exercises using you will get a good understanding of how you can use ACE for your application.

2.1.1. Goals

- Implement IPC mechanisms using the IPC SAP classes and the Acceptor/Connector pattern
- Utilize a Reactor in event demultiplexing and dispatching
- Implement thread-safe applications using the thread encapsulation class categories
- Identify appropriate ACE components

2.1.2. Audience

Software developers moving to distributed applications using ACE.

2.1.3. Duration

4 days

2.1.4. Prerequisites

Familiarity with the C++ language (including templates), software development in a Linux or Windows environment, and knowledge of the client-server architecture and network programming concepts.

2.1.5. Contents

- ACE Architecture and Components
- How to access Operating System services
- Overview of network programming interfaces
- Network programming using TCP and UDP classes in ACE
- Acceptor and Connector patterns
- Event demultiplexing with the Reactor
- Implementing event handlers for I/O, timers, and signals
- Thread management and synchronization
- Shared memory allocators and specialized local memory allocators
- Dynamic configuration with the Service Configurator
- Message Queues and Stream processing
- Logging and Debugging

2.1.6. Format

Lecture and programming exercises.

2.1.7. Material

Each student will get a print-out of all the sheets and a copy of C++NPv1 and C++NPv2, and a copy of the ACE Programmers Guide.

2.2. Introduction to CORBA

This one day introduction will provide an overview of CORBA.

2.2.1. Goals

- The benefits of distributed objects
- CORBA’s role in developing distributed applications
- To be able to determine when and where to apply CORBA
- The development trends in CORBA

2.2.2. Audience

Software Developers and Managers who are getting started with CORBA development

2.2.3. Duration

1 day

2.2.4. Prerequisites

Familiarity with to C++ and object-oriented concepts is desired.

2.2.5. Contents

- Distributed Objects
- The Object Management Group (OMG) Object Model
- Architecture of CORBA
- Interface Definition Language (IDL)
- The Object Request Broker (ORB)
- CORBA Services
- CORBA Frameworks
- Commercial tools for developing CORBA applications
- Comparison of CORBA to other distributed object standards

2.2.6. Format

Lectures.

2.2.7. Material

Each student will get a print-out of all the sheets.

2.3. CORBA Programming with C++

Our CORBA Programming with C++ course will learn you the concepts of CORBA and the IDL to C++ language mapping. Through lectures and a set of exercises using TAOX11 you will get a good understanding of how you can use CORBA for your application.

2.3.1. Goals

- Understand CORBA’s role in developing distributed applications
- Define CORBA interfaces using Interface Definition Language (IDL)
- Create CORBA clients and servers using C++
- Use the advanced features of the Portable Object Adapter (POA) in your applications

2.3.2. Audience

Software developers who will be developing distributed applications using CORBA.

2.3.3. Duration

4 days

2.3.4. Prerequisites

Non-trivial experience with C++ and familiarity with object-oriented concepts is required.

2.3.5. Contents

- What is CORBA?
- Interface Definition Language (IDL)
- CORBA Object Overview
- IDL to C++ Mapping
- Object Reference Details
- Parameter passing Rules - In, Out, Inout, Return
- Implementing Servants
- Managing Servants
- POA Details
- Request Routing Alternatives
- The Naming Service
- The Event Service
- Advanced Topics

2.3.6. Format

Lecture and programming exercises.

2.3.7. Material

Each student will get a print-out of all the sheets and a copy of Advanced CORBA Programming with C++ from Michi Henning & Stephen Vinoski.

2.4. CORBA Programming with C++11

Our CORBA Programming with C++11 course will learn you the concepts of CORBA and the IDL to C++11 language mapping. Through lectures and a set of exercises using TAOX11 you will get a good understanding of how you can use CORBA for your application.

2.4.1. Goals

- Understand CORBA's role in developing distributed applications
- Define CORBA interfaces using Interface Definition Language (IDL)
- Create CORBA clients and servers using C++11
- Use the advanced features of the Portable Object Adapter (POA) in your applications

2.4.2. Audience

Software developers who will be developing distributed applications using CORBA.

2.4.3. Duration

4 days

2.4.4. Prerequisites

Non-trivial experience with C++ and familiarity with object-oriented concepts is required.

2.4.5. Contents

- What is CORBA?
- Interface Definition Language (IDL)
- CORBA Object Overview
- IDL to C++11 Mapping
- Object Reference Details
- Implementing Servants
- Managing Servants
- POA Details
- Request Routing Alternatives
- The Naming Service
- Advanced Topics

2.4.6. Format

Lecture and programming exercises.

2.4.7. Material

Each student will get a print-out of all the sheets.

2.5. Advanced CORBA Programming with TAO

Our Advanced CORBA Programming with TAO course will learn you the advanced concepts of CORBA and the TAO specific features and configuration. Through lectures and a set of exercises using TAO you will get a good understanding of how you can use CORBA for your application.

2.5.1. Goals

- Extend and enhance your CORBA and C++ application development skills and techniques
- Use the Messaging and Real-Time CORBA features to take greater control of your CORBA application
- Explore the internals of TAO and how its different configurations affect your application
- Learn about the Notification and Real-Time Event Services

2.5.2. Audience

CORBA developers who wish to learn more about advanced CORBA features and TAO-specific features and configuration.

2.5.3. Duration

4 days

2.5.4. Prerequisites

Non-trivial CORBA and C++ programming experience is required.

2.5.5. Contents

- Configuring TAO Application
- Controlling Endpoints, Connections, and Protocols
- Building Multithreaded Applications with TAO
- CORBA QoS Policies
- Asynchronous Messaging
- Portable Interceptors
- Real-Time CORBA
- Using TAO’s Real-Time Event Service
- Notification Service
- TAO’s Implementation Repository

2.5.6. Format

Lecture and programming exercises.

2.5.7. Material

Each student will get a print-out of all the sheets and a copy of TAO Developers Guide. This course is not specific to any TAO distribution or version.

2.6. Component Based Development using AXCIOMA

This training explains the concepts behind AXCIOMA and through lectures and exercise you will experience the benefits of a component based approach.

2.6.1. Goals

- Understand the concepts of AXCIOMA
- Define and implement components using AXCIOMA
- Understand the concepts of connectors
- Integrating DDS through DDS4CCM
- Asynchronous invocations using AMI4CCM
- Using time triggers using TT4CCM
- Deploy your system using AXCIOMA

2.6.2. Audience

Software architects and developers who want to use AXCIOMA.

2.6.3. Duration

4 days

2.6.4. Prerequisites

Non-trivial experience with C++ and familiarity with object-oriented concepts is required.

2.6.5. Contents

- AXCIOMA introduction
- What is a component?
- Application Development Lifecycle
- Component Lifecycle explained

- Connectors and their place in CCM: CORBA4CCM, AMI4CCM, TT4CCM, and DDS4CCM
- Defining components using IDL3+
- Use BRIX11 APC to define, generate, and compile your project
- Basic tutorial using AXCIOMA
- Deployment and Configuration using AXCIOMA
- CCM and D&C modeling tools overview

2.6.6. Format

Lecture and programming exercises.

2.6.7. Material

Each student will get a print-out of all the sheets.

Chapter 3. Obtain your CORBA implementation

3.1. TAOX11

TAOX11 is a state of the art CORBA implementation which supports the IDL to C++11 language mapping.

TAOX11 is a free CORBA implementation which is developed by Remedy IT. TAOX11 can be obtained for free from Github.

3.2. R2CORBA

R2CORBA is the implementation of the Ruby CORBA Language mapping (RCLM) and is provided as open source.

3.3. TAO

The most recent TAO release can be downloaded from Github or Vanderbilt University. There you can download the latest minor, bug fix only, and micro version. Through a separate page you can download any older version.

The full version ships with generated GNU makefiles and Visual Studio project files. The source only packages lack the generated makefiles and project files and are therefor much smaller.

3.4. JacORB

JacORB is a 100% pure Java open source implementation of the OMG's CORBA standard and is fully compliant with the OMG IDL/Java language mapping version 2.3.

Chapter 4. TAOX11

4.1. Introduction

The IDL to C++ language mapping is hard to use. The IDL to C++11 resolves this completely, this language mapping is very easy to use. TAOX11 provides a CORBA implementation using the IDL to C++11 language mapping

This chapter explains the steps to create your first Hello world application using TAOX11. This example can be found in the distribution under `taox11/tests/hello`.

TAOX11 provides a powerful logging framework which is used by this getting started.

4.2. Define your application IDL

The first step is to define your application IDL. In the IDL specification you describe the interfaces the server delivers to its clients. Only the operations you define in IDL can be invoked by the client application. We want to implement a method that just returns the string send and another method to shutdown the server.

```
/// Put the interfaces in a module, to avoid global namespace pollution
module Test
{
  /// A very simple interface
  interface Hello
  {
    /// Return a simple string
    string get_string ();

    /// A method to shutdown the ORB
    oneway void shutdown ();
  };
};
```

Now you have defined your interfaces in IDL the client and server can be developed independent. The first step is to compile the IDL file using the ridlc compiler. ridlc converts the IDL into C++ classes that are the glue between your application code and the TAOX11 libraries. The compilation can be done using: `ridlc Hello.idl` After the compilation you now have `HelloC.{h,cpp}` and `HelloS.{h,cpp}`.

4.3. Implement the server

First we are going to develop the server part. For each interface we have to implement a C++ class that implements the in IDL defined methods. This class needs to include the generated HelloS.h header file.

```
#include "HelloS.h"
```

We derive from the base class generated by the IDL compiler

```
/// Implement the Test::Hello interface
class Hello final
  : public virtual CORBA::servant_traits<Test::Hello>::base_type
{
public:
  /// Constructor
  Hello (IDL::traits<CORBA::ORB>::ref_type orb);

  std::string get_string () override;

  void shutdown () override;

private:
  /// Use an ORB reference to shutdown the application.
  IDL::traits<CORBA::ORB>::ref_type orb_;
};
```

The implementation of this class is as below. First the constructor which received the ORB and stores it in a member variable which is used in the shutdown method.

```
#include "Hello.h"

Hello::Hello (IDL::traits<CORBA::ORB>::ref_type orb)
  : orb_ (std::move(orb))
{
}
```

The get_string method returns the hard coded string `Hello there!` to the client.

```
std::string
Hello::get_string (void)
{
  return "Hello there!";
}
```

With the shutdown method the client can shutdown the server.

```
void
Hello::shutdown ()
{
  this->orb_->shutdown (false);
}
```

Now we have implemented the class we need to implement the main of the server. We start with a regular main that uses its commandline arguments and provides a `try/catch` block to make sure we catch any exception.

```
int
main (int argc, char *argv[])
{
  try
    {
```

We now first initialize the ORB, retrieve the Root POA and POA Manager which we will use to activate our servant.

```
IDL::traits<CORBA::ORB>::ref_type orb =
  CORBA::ORB_init (argc, argv);

IDL::traits<CORBA::Object>::ref_type poa_object =
  orb->resolve_initial_references("RootPOA");

IDL::traits<PortableServer::POA>::ref_type root_poa =
  IDL::traits<PortableServer::POA>::narrow (poa_object);

if (!root_poa)
  {
    TAOX11_TEST_ERROR
      << "ERROR: IDL::traits<PortableServer::POA>::narrow (obj)"
      << "returned null object." << std::endl;
    return 1;
  }

IDL::traits<PortableServer::POAManager>::ref_type poa_manager =
  root_poa->the_POAManager ();
```

Now we create our servant and activate it

```
CORBA::servant_traits<Test::Hello>::ref_type hello_impl;
  CORBA::make_reference<Hello> (orb);

PortableServer::ObjectId id =
  root_poa->activate_object (hello_impl);

IDL::traits<CORBA::Object>::ref_type object =
  root_poa->id_to_reference (id);

IDL::traits<Test::Hello>::ref_type hello =
  IDL::traits<Test::Hello>::narrow (object);
```

We now write our IOR to a file on disk so that the client can find the server.

```
std::string ior = _orb->object_to_string (hello);

// Output the IOR to the ior_output_file
std::ofstream fos("server.ior");
if (!fos)
  {
    TAOX11_TEST_ERROR << "ERROR: failed to open file 'server.ior'" << std
::endl;
    return 1;
  }

fos << ior;
fos.close ();
```

Now we activate our POA Manager, at that moment the server accepts incoming requests and then run our ORB.

```
poa_manager->activate ();

orb->run ();
```

When the run method returns we print a message that we are ready and then destroy the RootPOA and the ORB.

```
root_poa->destroy (true, true);

orb->destroy ();
```

And we have a catch block to catch all exceptions and we use the ostream insertion support to print the exception information to the output.

```
  }
catch (const std::exception& ex)
{
  TAOX11_TEST_ERROR << "exception caught: " << ex.what ()
                    << std::endl;
  return 1;
}

return 0;
}
```

The server is now ready.

4.4. Implement the client

We implement the client application. When using TAOX11 the client is also written in C++ and

includes the generated HelloC.h header file.

```
#include "HelloC.h"
```

We start with a regular main that uses its commandline arguments and provides a `try/catch` block to make sure we catch any exception.

```
int
main (int argc, char *argv[])
{

  try
    {
```

We now first initialize the ORB and then do a `string_to_object` of the IOR file that server has written to disk. After this we do a `_narrow` to the derived interface.

```
IDL::traits<CORBA::ORB>::ref_type orb =
  CORBA::ORB_init (argc, argv);
IDL::traits<CORBA::Object>::ref_type tmp =
  orb->string_to_object("file://server.ior");
IDL::traits<Test::Hello>::ref_type hello =
  IDL::traits<Test::Hello>::narrow(tmp);
```

We now have to check whether we have a valid object reference or not. If we invoke an operation on a nil object reference we will cause an access violation.

```
if (!hello)
  {
    TAOX11_TEST_ERROR
      << "ERROR: IDL::traits<Test::Hello>::narrow (obj) "
      << "returned null object." << std::endl;
    return 1;
  }
```

Now we are sure we have a valid object reference, so we invoke the `get_string()` operation on the server. We have at this moment no clue how long this operation could take, it could return in micro seconds, it could take days, this all depends on the server.

```
std::string the_string = hello->get_string ();
```

And now we print the string to standard output.

```
TAOX11_TEST_INFO << "hello->get_string () returned "
                 << the_string << std::endl;
```

To let this example end itself gracefully we first shutdown the server and then destroy our own ORB.

```
hello->shutdown ();
orb->destroy ();
```

To make sure we see any exception we do have a catch statement catching these exceptions and printing the exception information in a readable format.

```
}
catch (const std::exception& ex)
  {
    TAOX11_TEST_ERROR << "exception caught: " << e.what ()
                      << std::endl;
    return 1;
  }

return 0;
}
```

4.5. Compile client and server

TAOX11 gets shipped together with a product called Make Project Creator (MPC). This tool is used by the TAO development group to generate all project files but can also be used by you as user to generate your own project files. The section below specifies the MPC file for this project which can be converted to project files for your environment. First we define a custom_only project that will compile the IDL file.

```
project(*idl): ridl_ostream_defaults {
  idlflags += -Sp
  IDL_Files {
    Hello.idl
  }
  custom_only = 1
}
```

Then we create a server and client project. The `after` will make sure the client and server are build after the idl project in case you are using an environment that supports parallel builds. In the MPC file you specify your dependencies and the files that must be compiled in the server and the client application.

```
project(*Server): taox11_server {
  after += *idl
  Source_Files {
    Hello.cpp
    server.cpp
  }
  Source_Files {
    HelloC.cpp
    HelloS.cpp
  }
  IDL_Files {
  }
}

project(*Client): taox11_client {
  after += *idl
  Source_Files {
    client.cpp
  }
  Source_Files {
    HelloC.cpp
  }
  IDL_Files {
  }
}
```

This MPC file is then used to generate the project files. For For this generation you will need perl 5.8 or higher on your system. For windows users we advice Active State Perl. Generating the project files and compiling the source code in a platform independent way can be done easily using the brix11 tooling that is part of the TAOX11 product:

```
brix11 gen build -- make
```

4.6. Run your application

To run this application you need two command prompts or consoles. In the first one you first start the server, normally it just starts and doesn't give any output. If you want to get some debugging output from the TAO libraries, add `-ORBDebugLevel 5` to the commandline arguments of the server. In the second console you now run the client, this will invoke the `get_string` call to the server, print the string it gets back and it then calls `shutdown` on the server.

Chapter 5. TAO

5.1. Introduction

This chapter explains the steps to create your first Hello world application using TAO. This example can be found in the distribution under `ACE_wrappers/TAO/tests/Hello`.

5.2. Define your application IDL

The first step is to define your application IDL. In the IDL specification you describe the interfaces the server delivers to its clients. Only the operations you define in IDL can be invoked by the client application. We want to implement a method that just returns the string send and another method to shutdown the server.

```
/// Put the interfaces in a module, to avoid global namespace pollution
module Test
{
  /// A very simple interface
  interface Hello
  {
    /// Return a simple string
    string get_string ();

    /// A method to shutdown the ORB
    oneway void shutdown ();
  };
};
```

Now you have defined your interfaces in IDL the client and server can be developed independent. The first step is to compile the IDL file using the TAO_IDL compiler. The TAO_IDL converts the IDL into C++ classes that are the glue between your application code and the TAO libraries. The compilation can be done using: `tao_idl Hello.idl` After the compilation you now have `HelloC.{h,cpp,inl}` and `HelloS.{h,cpp,inl}`. If you need to TIE approach you need to add the `-GT` flag to the invocation of TAO_IDL, this will create `HelloS_T.{h,cpp,inl}`.

5.3. Implement the server

First we are going to develop the server part. For each interface we have to implement a C++ class that implements the in IDL defined methods. This class needs to include the generated HelloS.h header file.

```
#include "HelloS.h"
```

We derive from the base class generated by the IDL compiler

```
/// Implement the Test::Hello interface
class Hello
  : public virtual POA_Test::Hello
{
public:
  /// Constructor
  Hello (CORBA::ORB_ptr orb);

  virtual char * get_string (void);

  virtual void shutdown (void);

private:
  /// Use an ORB reference to shutdown the application.
  CORBA::ORB_var orb_;
};
```

The implementation of this class is as below. First the constructor which received the ORB and duplicates it to a member variable which is used in the shutdown method.

```
#include "Hello.h"

Hello::Hello (CORBA::ORB_ptr orb)
  : orb_ (CORBA::ORB::_duplicate (orb))
{
}
```

The get_string method returns the hard coded string `Hello there!` to the client.

```
char *
Hello::get_string (void)
{
  return CORBA::string_dup ("Hello there!");
}
```

With the shutdown method the client can shutdown the server.

```
void
Hello::shutdown (void)
{
  this->orb_->shutdown (0);
}
```

Now we have implemented the class we need to implement the main of the server. We start with a regularly main that uses its commandline arguments and uses a `try/catch` block to make sure we catch any exception.

```
int
ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  try
    {
```

We now first initialize the ORB, retrieve the Root POA and POA Manager which we will use to activate our servant.

```
CORBA::ORB_var orb =
  CORBA::ORB_init (argc, argv);

CORBA::Object_var poa_object =
  orb->resolve_initial_references("RootPOA");

PortableServer::POA_var root_poa =
  PortableServer::POA::_narrow (poa_object.in ());

if (CORBA::is_nil (root_poa.in ()))
  ACE_ERROR_RETURN ((LM_ERROR,
                     " (%P|%t) Panic: nil RootPOA\n"),
                     1);

PortableServer::POAManager_var poa_manager =
  root_poa->the_POAManager ();
```

Now we create our servant and activate it

```
Hello *hello_impl = 0;
ACE_NEW_RETURN (hello_impl,
                Hello (orb.in ()),
                1);
PortableServer::ServantBase_var owner_transfer(hello_impl);

PortableServer::ObjectId_var id =
  root_poa->activate_object (hello_impl);

CORBA::Object_var object = root_poa->id_to_reference (id.in ());

Test::Hello_var hello = Test::Hello::_narrow (object.in ());
```

We now write our IOR to a file on disk so that the client can find the server. To get a real portable server application we are using ACE for the file access.

```
CORBA::String_var ior = orb->object_to_string (hello.in ());

// Output the IOR to the ior_output_file
FILE *output_file= ACE_OS::fopen ("server.ior", "w");
if (output_file == 0)
  ACE_ERROR_RETURN ((LM_ERROR,
                     "Cannot open output file for writing IOR: %s\n",
                     ior_output_file),
                     1);
ACE_OS::fprintf (output_file, "%s", ior.in ());
ACE_OS::fclose (output_file);
```

Now we activate our POA Manager, at that moment the server accepts incoming requests and then run our ORB.

```
poa_manager->activate ();

orb->run ();
```

When the run method returns we print a message that we are ready and then destroy the RootPOA and the ORB.

```
ACE_DEBUG ((LM_DEBUG, "(%P|%t) server - event loop finished\n"));

root_poa->destroy (1, 1);

orb->destroy ();
```

And we have a catch block to catch CORBA exceptions and we use the TAO specific `_tao_print_exception` to print the exception information to the output.

```
  }
catch (const CORBA::Exception&amp;amp; ex)
{
  ex._tao_print_exception ("Exception caught:");
  return 1;
}

return 0;
}
```

The server is now ready.

5.4. Implement the client

We implement the client application. When using TAO the client is also written in C++ and includes the generated HelloC.h header file.

```
#include "HelloC.h"
```

We start with a regularly ACE_TMAIN that uses its commandline arguments and uses a `try/catch` block to make sure we catch any exception.

```
int
ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{

  try
    {
```

We now first initialize the ORB and then do a `string_to_object` of the IOR file that server has written to disk. After this we do a `_narrow` to the derived interface.

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
CORBA::Object_var tmp = orb->string_to_object("file://server.ior");
Test::Hello_var hello = Test::Hello::_narrow(tmp.in ());
```

We now have to check whether we have a valid object reference or not. If we invoke an operation on a nil object reference we will cause an access violation.

```
if (CORBA::is_nil (hello.in ()))
   {
     ACE_ERROR_RETURN ((LM_DEBUG,
                        "Nil Test::Hello reference\n"),
                        1);
   }
```

Now we are sure we have a valid object reference, so we invoke the `get_string()` operation on the server. We have at this moment no clue how long this operation could take, it could return in micro seconds, it could take days, this all depends on the server.

```
CORBA::String_var the_string = hello->get_string ();
```

And now we print the string to standard output.

```
ACE_DEBUG ((LM_DEBUG, "(%P|%t) - string returned %C\n",
            the_string.in ()));
```

To let this example end itself gracefully we first shutdown the server and then destroy our own ORB.

```
hello->shutdown ();
orb->destroy ();
```

To make sure we see any CORBA exception we do have a catch statement catching these exceptions and printing the exception information in a readable format. Note that the `_tao_print_exception` is a TAO specific method.

```
}
catch (const CORBA::Exception&amp;amp; ex)
  {
    ex._tao_print_exception ("Exception caught:");
    return 1;
  }

return 0;
}
```

5.5. Compile client and server

TAO gets shipped together with a product called Make Project Creator (MPC). This tool is used by the TAO development group to generate all project files but can also be used by you as user to generate your own project files. The section below specifies the MPC file for this project which can be converted to project files for your environment. First we define a custom_only project that will compile the idl file.

```
project(*idl): taoidldefaults {
  IDL_Files {
    Hello.idl
  }
  custom_only = 1
}
```

Then we create a server and client project. The `after` will make sure the client and server are build after the idl project in case you are using an environment that supports parallel builds. In the MPC file you specify your dependencies and the files that must be compiled in the server and the client application.

```
project(*Server): taoserver {
  after += *idl
  Source_Files {
    Hello.cpp
    server.cpp
  }
  Source_Files {
    HelloC.cpp
    HelloS.cpp
  }
  IDL_Files {
  }
}

project(*Client): taoclient {
  after += *idl
  Source_Files {
    client.cpp
  }
  Source_Files {
    HelloC.cpp
  }
  IDL_Files {
  }
}
```

This MPC file is then used to generate the project files. For For this generation you will need perl 5.8 or higher on your system. For windows users we advice Active State Perl. Generating the project files for GNU make can be done with the following command:

```
$ACE_ROOT/bin/mwc.pl -type gnuace
```

On Windows, with Visual C++ 14, you can generate the solution and project files with MPC:

```
$ACE_ROOT/bin/mwc.pl -type vc14
```

MPC is capable of generating more types of project types, to see a list of possible project types use `$ACE_ROOT/bin/mwc.pl -help`

5.6. Run your application

To run this application you need two command prompts or consoles. In the first one you first start the server, normally it just starts and doesn't give any output. If you want to get some debugging output from the TAO libraries, add `-ORBDebugLevel 5` to the commandline arguments of the server. In the second console you now run the client, this will invoke the `get_string` call to the server, print the string it gets back and it then calls `shutdown` on the server.

Chapter 6. R2CORBA

6.1. Introduction

R2CORBA is a product developed by Remedy IT which makes it possible to implement a CORBA client or server using the Ruby programming language. Ruby is a dynamic open source programming language with a focus on simplicity and productivity. In the past there have been a few attempts to implement a full ORB in Ruby which in itself is a huge amount of work because of the large amount of features CORBA delivers. R2CORBA takes a different approach, we are using TAO as real ORB and this is then wrapped and made accessible for the Ruby programs.

6.2. Ruby CORBA mapping

For R2CORBA we had to create an CORBA language mapping. This is available from the OMG website.

6.3. Download R2CORBA

You can download R2CORBA for free from the Remedy IT website.

6.4. Define your IDL

As with any CORBA application we first have to define our IDL interfaces. We are going to implement the Hello world example using Ruby, so we define an interface with a `get_string()` method to retrieve a string and a `shutdown()` method to shutdown the server.

```
/// Put the interfaces in a module, to avoid global namespace pollution
module Test
{
  /// A very simple interface
  interface Hello
  {
    /// Return a simple string
    string get_string ();

    /// A method to shutdown the ORB
    /**
     * This method is used to simplify the test shutdown process
     */
    oneway void shutdown ();
  };
};
```

6.5. Implement a client

As part of the R2CORBA an IDL compiler for Ruby is delivered. Because Ruby is a powerful

scripting language it is not required to precompile your IDL file to Ruby code (which you would normally do when for example using C++). With R2CORBA you only have to specify which IDL files you want to use. You do this using the `implement` method of the CORBA module. This method will lookup the IDL file using the provided pathname and, like the Ruby `require` method, process a code module only once.

```
require 'r2tao'
CORBA.implement('Test.idl')
```

The first step is to initialize the ORB using ORB_init. To the ORB_init call you can pass an array of ORB initialization options (there is no manipulation of the argument array like with the C++ mapping).

```
orb = CORBA.ORB_init(["-ORBDebugLevel", 10], 'myORB')
```

We assume that the server has written an IOR file on disk, this is then used by the client program to get an object reference.

```
obj = orb.string_to_object("file://server.ior")
hello_obj = Test::Hello._narrow(obj)
```

Now that we have an object reference we can invoke the get_string() operation and print the content of the string.

```
the_string = hello_obj.get_string()

puts "string returned <#{the_string};>"
```

After this we invoke the shutdown() method on the server to let it shutdown and then we destroy our own ORB.

```
hello_obj.shutdown()

orb.destroy()
```

6.6. Implement a server

```
require 'r2tao'
CORBA.implement('Test.idl', {}, CORBA::IDL::SERVANT_INTF)
```

```
class MyHello < POA::Test::Hello
  def initialize(orb)
    @orb = orb
  end

  def get_string()
    ["Hello there!"]
  end

  def shutdown()
    @orb.shutdown
  end
end #of servant MyHello

orb = CORBA.ORB_init(["-ORBDebugLevel", 0], 'myORB')

obj = orb.resolve_initial_references('RootPOA')

root_poa = PortableServer::POA._narrow(obj)

poa_man = root_poa.the_POAManager

poa_man.activate

hello_srv = MyHello.new(orb)

hello_oid = root_poa.activate_object(hello_srv)

hello_obj = root_poa.id_to_reference(hello_oid)

hello_ior = orb.object_to_string(hello_obj)

open("server.ior", 'w') { |io|
  io.write hello_ior
}

orb.run
```

Chapter 7. TAO IDL Compiler

7.1. Generated Files

The TAO IDL compiler generates by default 6 files from each .idl file, optionally it can create 3 addition files. The file names are obtained by taking the IDL basename and appending the following suffixes (see the list of TAO's IDL compiler options on how to get different suffixes for these files:)

Client stubs, i.e., `*C.{h,cpp,inl}`. Pure client applications only need to #include and link with these files.

Server skeletons, i.e., `*S.{h,cpp,inl}`. Servers need to #include and link with these files.

Server skeleton templates, i.e., `*S_T.{h,cpp,inl}`. These are generated optionally using the -GT option. Some C++ compilers do not like template and non-template code in the same files, so TAO's IDL compiler generates these files separately.

TAO's IDL compiler creates separate **`.inl`** and `*S_T.{h,cpp,inl} files to improve the performance of the generated code. The .inl` files enable you to compile with inlining enabled or not, which is useful for trading off compiletime and runtime performance. Fortunately you only need to #include the client stubs declared in the `*C.h` file and the skeletons declared in the `*S.h` file in your code.

7.2. Environment Variables

TAO IDL supports the environment variables listed in the following table.

Table 1. TAO IDL Environment Variables

Variable	Usage
TAO_IDL_PREPROCESSOR	Used to override the program name of the preprocessor that TAO IDL uses
TAO_IDL_PREPROCESSOR_ARGS	Used to override the flags passed to the preprocessor that TAOIDL uses. This can be used to alter the default options for the preprocessor and specify things like include directories and how the preprocessor is invoked. Two flags that will always be passed to the preprocessor are -DIDL and -I..
TAO_ROOT	Used to determine where orb.idl is located
ACE_ROOT	Used to determine where orb.idl is located

Because TAO_IDL doesn't have any code to implement a preprocessor, it has to use an external one. For convenience, it uses a built-in name for an external preprocessor to call. During

compilation, this is how that default is set:

1. If the macro TAO_IDL_PREPROCESSOR is defined, then it will use that
2. Else if the macro ACE_CC_PREPROCESSOR is defined, then it will use that
3. Otherwise, it will use "cc"

And the same behavior occurs for the `TAO_IDL_PREPROCESSOR_ARGS` and `ACE_CC_PREPROCESSOR_ARGS` macros.

Case 1 is used by the Makefile on most machines to specify the preprocessor. Case 2 is used on Windows and platforms that need special arguments passed to the preprocessor (MVS, HPUX, etc.). And case 3 isn't used at all, but is included as a default case.

Since the default preprocessor may not always work when TAO_IDL is moved to another machine or used in cross-compilation, it can be overriden at runtime by setting the environment variables TAO_IDL_PREPROCESSOR and TAO_IDL_PREPROCESSOR_ARGS.

If ACE_ROOT or TAO_ROOT are defined, then TAO_IDL will use them to include the $(ACE_ROOT)/TAO/tao or $(TAO_ROOT)/tao directories. This is to allow TAO_IDL to automatically find `orb.idl` when it is included in an IDL file. TAO_IDL will display a warning message when neither is defined.

7.3. Operation Demuxing Strategies

The server skeleton can use different demuxing strategies to match the incoming operation with the correct operation at the servant. TAO's IDL compiler supports perfect hashing, binary search, and dynamic hashing demuxing strategies. By default, TAO's IDL compiler tries to generate perfect hash functions, which is generally the most efficient and predictable operation demuxing technique. To generate perfect hash functions, TAO's IDL compiler uses gperf, which is a general-purpose perfect hash function generator.

To configure TAO's IDL compiler to support perfect hashing please do the following:

1. Enable `ACE_HAS_GPERF` when building ACE and TAO. This macro has been defined for the platforms where gperf has been tested, which includes most platforms that ACE runs on
2. Build the gperf in $ACE_ROOT/apps/gperf/src. This build also leaves a copy/link of the gperf program at the $ACE_ROOT/bin directory
3. Set the environment variable $ACE_ROOT appropriately or add $ACE_ROOT/bin to your search path
4. Use the -g option for the TAO IDL compiler or set your search path accordingly to install gperf in a directory other than $ACE_ROOT/bin

Note that if you can't use perfect hashing for some reason the next best operation demuxing strategy is binary search, which can be configured with the option described in the following table.

Table 2. TAO_IDL Operation Demuxing Strategies

Option	Usage
-H perfect_hash	To specify the IDL compiler to generate skeleton code that uses perfect hashed operation demuxing strategy, which is the default strategy. Perfect hashing uses gperf program, to generate demuxing methods
-H dynamic_hash	To specify the IDL compiler to generate skeleton code that uses dynamic hashed operation demuxing strategy.
-H binary_search	To specify the IDL compiler to generate skeleton code that uses binary search based operation demuxing strategy
-H linear_search	To specify the IDL compiler to generate skeleton code that uses linear search based operation demuxing strategy. Note that this option is for testing purposes only and should not be used for production code since it's inefficient

7.4. Collocation Strategies

TAO_IDL can generate collocated stubs using two different collocation strategies. It also allows you to supress/enable the generation of the stubs of a particular strategy. To gain great flexibility at runtime, you can gereate stubs for both collocation strategies (using both '-Gp' and '-Gd' flags at the same time) and defer the determination of collocation strategy until runtime. On the other hand, if you want to minimize the footprint of your program, you might want to pre-determine the collocation strategy you want and only generate the right collocated stubs (or not generating any at all using both '-Sp' and '-Sd' flags at the same time if it's a pure client.) See the collocation paper for a detail discussion on the collocation support in TAO. Note that there is a bug in TAO 1.5.x which causes a crash when you select in runtime a collocation strategy for which the collocation strategy hasn't been generated by the IDL compiler (2241)

7.5. Output File options

With TAO_IDL you can control the filenames that are generated. An overview of the available options are listed in the next table.

Table 3. TAO IDL Output File Options

Option	Usage	Default
-o	Specify the output directory where all the IDL compiler generated files are to be put	Current directory
-oS	Same as -o option but applies only to generated **S.** files	Value of -o option
-oA	Same as -o option but applies only to generated **A.** files	Value of -o option
-hc	Client's header file name ending	C.h
-hs	Server's header file name ending	S.h

Option	Usage	Default
-hT	Server's template header file name ending	S_T.h
-cs	Client stub's file name ending	C.cpp
-ci	Client inline file name ending	C.inl
-ss	Server skeleton file name ending	S.cpp
-sT	Server template skeleton file name ending	S_T.cpp
-si	Server inline skeleton file name ending	S.inl
-GIh	Servant implementation header file name ending	I.h
-GIs	Servant implementation skeleton file name ending	I.cpp

7.6. Controlling code generation

TAO_IDL delivers a set of options with which you can control the code generation. We have options to generate additional code parts as listed in this table, or to suppress parts that we generate by default but are not required for some applications as listed in this table.

Table 4. TAO_IDL Additional flags

Option	Usage
-GT	Enable generation of the TIE classes, and the **S_T.** files that contain them
-GA	Generate type codes and Any operators in *A.h and *A.cpp. Decouples client and server decisions to compile and link TypeCode- and Any-related code, which is generated in *C.h and *C.cpp by default. If -Sa or -St also appear, then an empty *A.h file is generated.
-GC	Generate AMI stubs, "sendc_" methods, reply handler stubs, etc
-GH	Generate AMH stubs, skeletons, exception holders, etc
-Gp	Generated collocated stubs that use Thru_POA collocation strategy (default enabled)
-Gd	Generated collocated stubs that use Direct collocation strategy
-Gsp	Generate client smart proxies
-Gt	Generate optimized TypeCodes
-GX	Generate empty A.h file. Used by TAO developers for generating an empty A.h file when the -GA option can't be used. Overridden by -Sa and -St.

Option	Usage
-Guc	Generate uninlined constant if defined in a module. Inlined (assigned a value in the C++ header file) by default, but this causes a problem with some compilers when using pre-compiled headers. Constants declared at global scope are always generated inline, while those declared in an interface or a valuetype never are - neither case is affected by this option
-Gse	Generate explicit export of sequence's template base class. Occasionally needed as a workaround for a bug in Visual Studio (.NET 2002, .NET 2003 and Express 2005) where the template instantiation used for the base class isn't automatically exported
-Gos	Generate ostream operators for IDL declarations. Can be useful for exception handling and debugging
-Gce	Generate code targeted at CORBA/e
-Gmc	Generate code targeted at Minimum CORBA

TAO IDL supports the following suppression flags. Note that all the file suppresion options don't check whether something is generated in the file. It just suppresses it without looking at any possible contents

Table 5. TAO_IDL Suppression flags

Option	Usage
-Sa	Suppress generation of the Any operators
-Sal	Suppress generation of the Any operators for local interfaces only
-Sp	Suppress generation of collocated stubs that use Thru_POA collocation strategy
-Sd	Suppress generation of collocated stubs that use Direct collocation strategy (default)
-St	Suppress generation of typecodes. Also suppresses the generation of the Any operators, since they need the associated typecode
-Sm	Suppress C++ code generation from CCM 'implied' IDL. This code generation is achieved by default using a 'preprocessing' visitor that modified the AST and is launched just before the code generating visitors. There is a new tool in CIAO that converts the entire IDL file into one containing explicit declarations of the implied IDL types. For such a file, we don't want the preprocessing visitor to be launched, so this command line option will suppress it
-SS	Suppress generation of the skeleton implementation and inline file
-Sci	Suppress generation of the client inline file
-Scc	Suppress generation of the client stub file
-Ssi	Suppress generation of the server inline file
-Ssc	Suppress generation of the server skeleton file

Option	Usage
-Sorb	Suppress generation of the ORB.h include. This option is useful when regenerating pidl files in the core TAO libs to prevent cyclic includes

7.7. Backend options

This table described the backend options. These options have to be passed to the IDL compiler in the format:

```
-Wb,optionlist
```

The option list is a comma-separated list of the listed backend options.

Table 6. TAO_IDL Backend Options

Option	Usage
Option	Usage
skel_export_macro=macro_name	The compiler will emit macro_name right after each class or extern keyword in the generated skeleton code (S files,) this is needed for Windows, which requires special directives to export symbols from DLLs, usually the definition is just a space on unix platforms.
skel_export_include=include_path	The compiler will generate code to include include_path at the top of the generated server header, this is usually a good place to define the server side export macro.
stub_export_macro=macro_name	The compiler will emit macro_name right after each class or extern keyword in the generated stub code, this is needed for Windows, which requires special directives to export symbols from DLLs, usually the definition is just a space on unix platforms.
stub_export_include=include_path	The compiler will generate code to include include_path at the top of the client header, this is usually a good place to define the export macro.

Option	Usage
anyop_export_macro=macro_name	The compiler will emit macro_name before each Any operator or extern typecode declaration in the generated stub code, this is needed for Windows, which requires special directives to export symbols from DLLs, usually the definition is just a space on unix platforms. This option works only in conjuction with the -GA option, which generates Any operators and typecodes into a separate set of files.
anyop_export_include=include_path	The compiler will generate code to include include_path at the top of the anyop file header, this is usually a good place to define the export macro. This option works in conjuction with the -GA option, which generates Any operators and typecodes into a separate set of files.
export_macro=macro_name	This option has the same effect as issuing -Wb,skel_export_macro=macro_name -Wb,stub_export_macro=macro_name -Wb,anyop_export_macro=macro_name. This option is useful when building a DLL containing both stubs and skeletons.
export_include=include_path	This option has the same effect as specifying -Wb,stub_export_include=include_path -Wb,skel_export_include=include_path -Wb,anyop_export_include=include_path. This option goes with the previous option to build DLL containing both stubs and skeletons.
pch_include=include_path	The compiler will generate code to include include_path at the top of all TAO IDL compiler generated files. This can be used with a precompiled header mechanism, such as those provided by Embarcadero C++ Builder or MSVC++.
obv_opt_accessor	The IDL compiler will generate code to optimize access to base class data for valuetypes.

Option	Usage
pre_include=include_path	The compiler will generate code to include include_path at the top of the each header file, before any other include statements. For example, ace/pre.h, which pushes compiler options for the Embarcadero C++ Builder and MSVC++ compilers, is included in this manner in all IDL-generated files in the TAO libraries and CORBA services.
post_include=include_path	The compiler will generate code to include include_path at the bottom of the each header file. For example, ace/post.h, which pops compiler options for the Embarcadero C++ Builder and MSVC++ compilers, is included in this manner in all IDL-generated files in the TAO libraries and CORBA services.
include_guard=define	The compiler will generate code the define in the C.h file to prevent users from including the generated C.h file. Useful for regenerating the pidl files in the archive.
safe_include=file	File that the user should include instead of this generated C.h file. Useful for regenerating the pidl files in the archive.
unique_include=file	File that the user should include instead of the normal includes in the C.h file. Useful for regenerating the *_include pidl files in the archive.

7.8. Other options

Besides all the options listed in the previous sections we do have a set of other options. These are listed in this table.

Table 7. TAO_IDL Other flags

Option	Usage
-u	The compiler prints out the options that are given below and exits clean
-V	The compiler printouts its version and exits
-E	Invoke only the preprocessor
-Wp,option_list	Pass options to the preprocessor.
-d	Causes output of a dump of the AST

Option	Usage
-Dmacro_definition	It is passed to the preprocessor
-Umacro_name	It is passed to the preprocessor
-Iinclude_path	It is passed to the preprocessor
-Aassertion	It is passed to the preprocessor
-Yp,path	Specifies the path for the C preprocessor
-in	To generate #include statements with <>'s for the standard include files (e.g. tao/corba.h) indicating them as non-changing files
-ic	To generate #include statements with ""s for changing standard include files (e.g. tao/corba.h)
-t	Temporary directory to be used by the IDL compiler. Unix: use environment variable TEMPDIR if defined, else use /tmp/. Windows NT/2000/XP: use environment variable TMP or TEMP if defined, else use the Windows directory
-Cw	Output a warning if two identifiers in the same scope differ in spelling only by case (default is output of error message). This option has been added as a nicety for dealing with legacy IDL files, written when the CORBA rules for name resolution were not as stringent.
-Ce	Output an error if two identifiers in the same scope differ in spelling only by case (default)

Chapter 8. TAO libraries

As part of the subsetting effort to reduce footprint of applications using TAO, we have created different libraries that house various CORBA features, such the POA and DynamicAny. This design helps minimize application footprint, only linking in features that are required. However, applications must link in the libraries they need. It is possible to load most of these libraries dynamically using the ACE Service Configurator framework, though this will not work for statically linked executables. Linking the necessary libraries with your application is therefore the most straightforward way to get the features you need.

Here we outline the list of libraries in TAO core with the list of MPC projects that can be used by the application to get all the required libraries linked into the application. The library names in table below are base names which can get a prefix and postfix depending on your platform and configuration. For example UNIX based systems have mostly a `lib` prefix and `.so` postfix. Windows systems have a slightly different naming convention, e.g., the PortableServer library is named as PortableServerd.lib and PortableServerd.dll. But for the naming conventions used on different platforms, the contents of the libraries and the dependencies outlined below are the same.

Table 8. List of CORE Libraries in TAO

Name of the Library	Feature	MPC project to use
TAO	All the core features for a client and server side ORB. The list includes support for IIOP, invocation framework, wait strategies for transports, leader-follower framework, thread pools and thread-per-connection framework, CORBA Policy framework, CDR framework, etc	taoclient
TAO_AnyTypeCode	Library with all the TypeCode and Any support. If you use the anytypecode base project the IDL compiler flags -Sa and -St are removed from the default idl flags.	anytypecode
TAO_BiDirGIOP	Support for BiDirectional GIOP as outlined by the CORBA spec. Please see $TAO_ROOT/tests/BiDirectional for a simple test case of this feature. Applications need to ` #include "tao/BiDir_GIOP/BiDirGIOP.h" ` within their code to get this feature.	bidir_giop
TAO_CodecFactory	Support for CodecFactory as outlined by the CORBA spec. Please see $TAO_ROOT/tests/Codec for a simple test case of this feature. Applications need to ` #include "tao/CodecFactory/CodecFactory.h" ` within their code to get this feature.	codecfactory

Name of the Library	Feature	MPC project to use
TAO_Domain	Support for server side skeletons for the DomainManager interface.	No base projects available
TAO_DynamicAny	Support for DynamicAny. Please see $TAO_ROOT/tests/DynAny_Test for an example of how to access and use this library. Applications have to ` #include "tao/DynamicAny/DynamicAny.h" ` to get the right symbols.	dynamicany
TAO_EndpointPolicy	Support for the TAO-specific Endpoint Policy. This is used to set up constraints on endpoints placed in IORs. The endpoint policy is applied to a POAManager via the POAManagerFactory and affects all POAs associated with that manager. Examples of use are in $TAO_ROOT/tests/POA/EndpointPolicy. Applications have to ` #include "tao/EndpointPolicy/EndpointPolicy.h" ` to get the right symbols.	endpointpolicy
TAO_DynamicInterface	Support for DII and DSI invocations. Applications have to ` #include "tao/DynamicInterface/Dynamic_Adapter_Impl.h " ` to get the right symbols.	dynamicinterface
TAO_IFR_Client	Support for client/stub side interfaces for InterfaceRepository applications. Applications have to ` #include "tao/IFR_Client/IFR_Client_Adapter_Impl.h" ` to get the right symbols.	ifr_client
TAO_ImR_Client	Support for applications that want to register itself to the Implementation Repository. Applications have to ` #include "tao/ImR_Client/ImR_Client.h" ` to get the right symbols.	imr_client
TAO_IORInterceptor	Support for IORInterceptor. The portable server library depends on the IORInterceptor library. Applications have to ` #include "tao/IORInterceptor/IORInterceptor_Adapter_Fa ctory_Impl.h" ` to get the right symbols.	iorinterceptor

Name of the Library	Feature	MPC project to use
TAO_IORManipulation	Support for IOR manipulation. The interfaces offered provide operations to create and multi-profile IOR's and other related utilities. Applications have to ` #include "tao/IORManipulation/IORManip_Loader.h" ` to get the right symbols.	iormanip
TAO_IORTable	Any TAO server can be configured as an corbaloc agent. Such agents forward requests generated using a simple ObjectKey in a corbaloc specification to the real location of the object. In TAO we implement this feature by dynamically (or statically) adding a new Object Adapter to the ORB, that handles any sort of request. This feature is placed in this library. Applications have to ` #include "tao/IORTable/IORTable.h" ` to get the right symbols.	iortable
TAO_Messaging	Support for AMI and CORBA policies such as RoundtripTimeout and ConnectionTimeout are placed in this library. Applications have to ` #include "tao/Messaging/Messaging.h" ` to get the rightsymbols.	messaging
TAO_ObjRefTemplate	Support for Object Reference Template specification. The portable server library depends on this library.	objreftemplate
TAO_PI	Support for Portable Interceptors. This library is automagically loaded by the ORB when the application uses the PolicyFactory or ORBInitializer . Just linking this library should be sufficient to get all the features that are required to write applications using portable interceptors.	pi
TAO_PortableServer	Support for POA. This library is automagically loaded by the ORB when the application calls resolve_initial_references ("RootPOA"); Just linking this library should be sufficient to get all the features that are required to write powerful servers.	taoserver
TAO_RTCORBA	Support for RTCORBA client side features. Applications are required to ` #include "tao/RTCORBA/RTCORBA.h" ` to get the required symbols for linking. Support in this library is complaint with RTCORBA 1.0 spec.	rt_client

Name of the Library	Feature	MPC project to use
TAO_RTPortableServer	Support for RTCORBA server side features. Applications are required to ` #include "tao/RTPortableServer/RTPortableServer.h" ` to get the required symbols for linking. Support in this library is complaint with RTCORBA 1.0 spec.	rt_server
TAO_RTScheduling	Support for RTCORBA 1.2 features. Applications are required to ` #include "tao/RTScheduling/RTScheulding.h" ` to get the required symbols for linking. Support in this library is complaint with RTCORBA 1.2 spec.	rtscheduling
TAO_SmartProxies	Support for Smartproxies.	smart_proxies
TAO_Strategies	Support for advanced resource options for the ORB that have been strategized into this library. Advanced resource categories include new transport protocols, additional reactors, connection purging strategies etc. Applications should ` #include "tao/Strategies/advanced_resources.h" `.	strategies
TAO_TypeCodeFactory	Support for TypeCodeFactory interface.	typecodefactory
TAO_Utils	Helper methods for that are useful for writing portable, exception safe application code.	utils
TAO_Valuetype	Support for object by value (OBV). Portable server and messaging depends on this library	valuetype
TAO_CSD_Framework	Support framework for Custom Servant Dispatching (CSD) feature. The CSD_ThreadPool depends on this library	csd_framework
TAO_CSD_ThreadPool	Support for ThreadPool Custom Servant Dispatching (CSD) Strategy. This library can be loaded statically or dynamically. Applications are required to ` #include "tao/CSD_ThreadPool/CSD_ThreadPool.h" ` for static loading and provide service configuration file for dynamic loading.	csd_threadpool

Name of the Library	Feature	MPC project to use
TAO_TC	Support for TAO::Transport::Current - a generic fremework for aplications that need access to statistical information about the currently used Transport. This library can be loaded statically or dynamically. Applications are required to ` #include "tao/TransportCurrent/Transport_Current.h" ` for static loading.	tc
TAO_TC_IIOP	Support for TAO::Transport::IIOP::Current - an IIOP-specific plug-in for Transport::Current. This library can be loaded statically or dynamically. Applications are required to ` #include "tao/TransportCurrent/IIOP_Transport_Current.h" ` for static loading. Depends on libTAO_TC.so.	tc_iiop
TAO_Compression	Support for Compression. This library can be loaded statically or dynamically. Applications are required to ` #include "tao/Compression/Compression.h" ` for static loading.	compression
TAO_ZlibCompressor	Support for Zlib Compression. This library can be loaded statically or dynamically. Applications are required to ` #include "tao/Compression/zlib/ZlibCompressor.h" ` for static loading.	zlibcompressor

Chapter 9. Compression

Starting with TAO 1.5.5 the compression library exists. With this library it is possible to compress and decompress application data using pluggable compressors. This library is the first step in the development of Zipped IOP (ZIOP) which adds the ability that the ORB compresses all application data transparently that is send over a remote connection.

To be able to use compression a compressor should be available. For being able to use a compressor the compressor factory must be registered with the ORB. The compressor factory creates the compressors that can be used to (un)compress the data. As part of the TAO distribution a zlib compressor gets shipped, other compressor factories can be added by application developers.

9.1. Using compression

The include for the Compression library that must be used in the application code is as following.

```
#include "tao/Compression/Compression.h"
```

Then you have to include the compressor factories that are going to be used. The default zlib compressor factory can be included as following.

```
#include "tao/Compression/zlib/ZlibCompressor_Factory.h"
```

As in a normal CORBA application you first have to initialise the ORB.

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
```

Then you have to retrieve the CompressionManager using `resolve_initial_references()`.

```
CORBA::Object_var compression_manager =
  orb->resolve_initial_references("CompressionManager");

Compression::CompressionManager_var manager =
  Compression::CompressionManager::_narrow (compression_manager.in ());

if (CORBA::is_nil(manager.in ()))
  ACE_ERROR_RETURN ((LM_ERROR,
                     " (%P|%t) Panic: nil compression manager\n"),
                     1);
```

The compression manager has no compressors by default, you have to register the compressor factories that need to be available to your application.

```
Compression::CompressorFactory_ptr compressor_factory;

ACE_NEW_RETURN (compressor_factory, TAO::Zlib_CompressorFactory (), 1);

Compression::CompressorFactory_var compr_fact = compressor_factory;
manager->register_factory(compr_fact.in ());
```

Now all the setup has been done. When you need a compression you need to retrieve a compressor. The number passed into the get_compresor method is the id of the compressor you want. These predefine id's are listed in the following table.

Table 9. Compressor Ids

Id	Compressor
Compression::COMPRESSORID_GZIP	gzip
Compression::COMPRESSORID_PKZIP	pkzip
Compression::COMPRESSORID_BZIP2	bzip2
Compression::COMPRESSORID_ZLIB	zlib
Compression::COMPRESSORID_LZMA	lzma
Compression::COMPRESSORID_LZOP	lzo
Compression::COMPRESSORID_RZIP	rzip
Compression::COMPRESSORID_7X	7x
Compression::COMPRESSORID_XAR	xar

```
Compression::Compressor_var compressor = manager->get_compressor (
Compression::COMPRESSORID_ZLIB);
```

A compressor is capable of compression Compression::Buffer as data which can contain any data as byte array. When compression data you should pass in an out sequence to put the compressed data in. If you want to set a safe size, take the length of the original sequence and multiple it with 1.10, this safe size can be dependent on the compressor you are using. At the moment the size is not large enough a `Compression::CompressionException` will be thrown.

```
Compression::Buffer myout;
myout.length ((CORBA::ULong)(mytest.length() * 1.1));

compressor->compress (mytest, myout);
```

To decompress that data you pass in the compressed data and a second `Compression::Buffer` that can be used to put the decompressed data in, this `Compression::Buffer` must have a length large enough to contain the decompressed data. At the moment then second `Compression::Buffer` is not large enough a

`Compression::CompressionException` will be thrown. The compressed `Compression::Buffer` doesn't contain the size of the original data, if you need this when decompressing you have to transfer it to the function doing decompression yourself.

```
Compression::Buffer decompress;
decompress.length (1024);

compressor->decompress (myout, decompress);
```

Compression application types can be done using an `Any` as intermediate datatype. The `Any` can then be converted to a `OctetSeq` using the Codec (short for coder/decoder) support of CORBA. For information how to use the Codec see the related chapter.

9.2. Implementing your own compressor

As application developer you can add your own custom compressor. Adding a compressor will require you implement two classes, the CompressorFactory and the Compressor itself.

The CompressorFactory is capable of creating a compressor for a given compression level. To make the implementation of the CompressorFactory easier TAO delivers the CompressorFactory base class that stores common functionality.

```
class My_CompressorFactory : public ::TAO::CompressorFactory
{
  public:
    My_CompressorFactory (void);

    virtual ::Compression::Compressor_ptr get_compressor (
        ::Compression::CompressionLevel compression_level);
  private:
     ::Compression::Compressor_var compressor_;
};
```

First, the constructor. This is easy, we pass our compressor id to the base class and initialize our member to nil. The compressor id must be unique for each compression algoritm.

```
My_CompressorFactory::My_CompressorFactory (void) :
  ::TAO::CompressorFactory (12),
  compressor_ (::Compression::Compressor::_nil ())
{
}
```

The factory method that must be implemented is the get_compressor method. For simplicity we ignore the compression_level, we just have one compressor instance for all levels.

```
::Compression::Compressor_ptr
Zlib_CompressorFactory::get_compressor (
    ::Compression::CompressionLevel compression_level)
{
  if (CORBA::is_nil (compressor_.in ()))
    {
      compressor_ = new ZlibCompressor (compression_level, this);
    }

  return ::Compression::Compressor::_duplicate (compressor_.in ());
}
```

The CompressorFactory is now ready and we start to implement the Compressor itself. For simplifying the implementation we use the BaseCompressor helper base class. Besides the constructor we have to implement the compress and decompress methods

```
class MyCompressor : public ::TAO::BaseCompressor
{
  public:
    MyCompressor (::Compression::CompressionLevel compression_level,
                  ::Compression::CompressorFactory_ptr compressor_factory);

    virtual void compress (
        const ::Compression::Buffer &amp;source,
        ::Compression::Buffer &amp;target);

    virtual void decompress (
        const ::Compression::Buffer &amp;source,
        ::Compression::Buffer &amp;target);
};
```

The constructor just passes the values to its base, this compressor is very easy, it doesn't need to store any additonal data itself.

```
MyCompressor::MyCompressor (
  ::Compression::CompressionLevel compression_level,
  ::Compression::CompressorFactory_ptr compressor_factory) :
    BaseCompressor (compression_level, compressor_factory)
{
}
```

Then the compress method, we need to compress the data from the source into the target. At the moment compression fails we must throw a `Compression::CompressionException` exception.

```
void
MyCompressor::compress (
    const ::Compression::Buffer &amp;source,
    ::Compression::Buffer &amp;target
  )
{
  // do compression
}
```

The decompress method should do the opposite work of the compress method. At the moment decompression fails then also a `Compression::CompressionException` must be thrown.

```
void
MyCompressor::decompress (
  const ::Compression::Buffer &amp;source,
  ::Compression::Buffer &amp;target)
{
  // do decompression
}
```

If you have implemented a compressor, consider contributing that back to the TAO distribution so that other applications can also benefit from this compressor.

Chapter 10. Using the TAO::Transport::Current Feature

10.1. Scope and Context

In TAO, it is just too hard to obtain statistical or pretty much any operational information about the network transport which the ORB is using. While this is a direct corollary of the CORBA's design paradigm which mandates hiding all this hairy stuff behind non-transparent abstractions, it also precludes effective ORB and network monitoring.

The Transport::Current feature intends to fill this gap by defining a framework for developing a wide range of solutions to this problem. It also provides a basic implementation for the most common case - the IIOP transport.

By definition, transport-specific information is available in contexts where the ORB has selected a Transport:

- Within Client-side interception points
- Within Server-side interception points
- Inside a Servant up-call

The implementation is based on a generic service-oriented framework, implementing the TAO::Transport::Current interface. It is an optional service, which can be dynamically loaded. This service makes the Transport::Current interface available through `CORBA::ORB::resolve_initial_references()`. The basic idea is that whenever a Transport is chosen by the ORB, the Transport::Current (or a derivative) will have access to that instance and be able to provide some useful information.

10.2. Programmer's Reference

Consider the following IDL interface, describing a Factory for producing `TAO::Transport::Traits` instance, which represents transport-specific data.

```
#include <IOP.pidl>
#include <TimeBase.pidl>

module TAO
{
  /// A type used to represent counters
  typedef unsigned long long CounterT;

  module Transport
  {
    /// Used to signal that a call was made within improper invocation
    /// context.  Also, this exception is thrown if no Transport has
    /// been selected for the current thread, for example in a
    /// collocated invocation.

    exception NoContext
    {
    };

    // The primary interface, providing access to Transport
    // information, available to the current thread.

    local interface Current
    {
      /// Transport ID, unique within the process.
      readonly attribute long id raises (NoContext);

      /// Bytes sent/received through the transport.
      readonly attribute CounterT bytes_sent raises (NoContext);
      readonly attribute CounterT bytes_received raises (NoContext);

      /// Messages (requests and replies) sent/received using the current
      /// protocol.
      readonly attribute CounterT messages_sent raises (NoContext);
      readonly attribute CounterT messages_received raises (NoContext);

      /// The absolute time (miliseconds) since the transport has been
      /// open.
      readonly attribute TimeBase::TimeT open_since raises (NoContext);
    };
  };
};
```

As an example of a specialized Transport::Current is the Transport::IIOP::Current, which derives from Transport::Current and has an interface, described in the following IDL:

```
#include "TC.idl"

/// Provide a forward reference for the SSLIOP::Current
module SSLIOP
{
  interface Current;
};


module TAO
{
  module Transport
  {
    module IIOP
    {
      // The primary interface, providing access to IIOP-specific
      // transport information, if it is indeed an IIOP (-like) transport
      // that has been selected.

      local interface Current : TAO::Transport::Current
      {
        /// Remote host
        readonly attribute string remote_host raises (NoContext);

        /// Remote port Using long (signed) type to better accomodate
        /// the Java mapping, which has no support for unsigned values
        readonly attribute long remote_port raises (NoContext);

        /// Local host
        readonly attribute string local_host raises (NoContext);

        /// Local port
        readonly attribute long local_port raises (NoContext);

        /// If this is a "secure" transport, this method will give you
        /// the corresponding SSLIOP::Current
        readonly attribute ::SSLIOP::Current ssliop_current raises
(NoContext);
      };
    };
  };
};
```

10.3. User’s Guide

The TAO::Transport::Current can be used as a base interface for a more specialized TAO::Transport::X::Current. It is not required, however that a more specialized Current inherits from it.

Typical, generic usage is shown in the $TAO_ROOT/orbsvcs/tests/Transport_Current/Framework test:

```
// Get the Current object.
::CORBA::Object_var tcobject =
  orb->resolve_initial_references ("TAO::Transport::Current");

::TAO::Transport::Current_var tc =
    ::TAO::Transport::Current::_narrow (tcobject.in ());

if (CORBA::is_nil (tc.in ()))
  {
    ACE_ERROR ((LM_ERROR,
                ACE_TEXT ("(%P|%t) client - ERROR: Could not resolve ")
                ACE_TEXT ("TAO::Transport::Current object.\n")));

    throw ::CORBA::INTERNAL ();
  }
```

Another example is available from the $TAO_ROOT/tests/TransportCurrent/IIOP test. This fragment shows how to obtain transport-specific information.

```
// Get the specific Current object.
CORBA::Object_var tcobject =
  orb->resolve_initial_references ("TAO::Transport::IIOP::Current");

Transport::IIOP::Current_var tc =
  Transport::IIOP::Current::_narrow (tcobject.in ());

if (CORBA::is_nil (tc.in ()))
  throw ::CORBA::INTERNAL ();

::CORBA::String_var rhost (tc->remote_host ());
::CORBA::String_var lhost (tc->local_host ());
::CORBA::Long id = tc->id ();
::TAO::CounterT bs = tc->bytes_sent ();
::TAO::CounterT br = tc->bytes_received ();
::TAO::CounterT rs = tc->messages_sent ();
::TAO::CounterT rr = tc->messages_received ();
```

10.4. Configuration, Bootstrap, Initialization and Operation

To use the Transport Current features the framework must be loaded through the Service Configuration framework. For example, using something like this:

```
dynamic TAO_Transport_Current_Loader Service_Object *
  TAO_TC:_make_TAO_Transport_Current_Loader() ""
```

The Transport_Current_Loader service uses an ORB initializer to register the `TAO::Transport::Current` name in a way that allows it to be resolved via `CORBA::ORB::resolve_initial_references()`. The implementation is the

TAO::Transport::Current_Impl class.

A transport-specific Traits_Factory objects are loaded like this:

```
dynamic TAO_Transport_IIOP_Current_Loader Service_Object *
  TAO_TC_IIOP:_make_TAO_Transport_IIOP_Current_Loader() ""
```

Note that any number of transport-specific Current interfaces may be available at any one time.

Whenever a Transport::Current method is invoked, a pointer to the currently selected Transport instance must be accessible through Thread Specific Storage (TSS). For each thread, this is managed by modifying the TAO classes, instances of which are created on the stack during request/response processing.

10.5. Implementation and Required Changes

The primary implementation is predicated upon usage of thread specific storage (TSS) and the guarantees C++ provides for calling the constructor and the destructor of automatic (stack-based) objects. Some existing objects, used in TAO will have to be modified and the necessary changes, both for client and the server side are detailed below.

10.6. Client Side: Sending Requests or Replies

The Profile_Transport_Resolver instance contains the reference to the Transport, which is the TAO implementation structure that is needed to extract any protocol-specific information. An instance of Profile_Transport_Resolver lives on the stack, starting inside a call to Invocation_Adapter::invoke_remote_i(), or LocateRequest_Invocation_Adapter::invoke(). In the case of collocated invocations no such object is created.

It is then passed around the calls that follow, except for the calls to the following Invocation_Base methods: send_request_interception(), receive_other_interception(), receive_reply_interception(), handle_any_exception(), handle_all_exception();

Note that these in turn call the client-side interception points and that is where information about the transport will be needed. In order to make the transport information accessible inside those methods, we changed Profile_Transport_Resolver and the TAO_ServerRequest classes to incorporate an additional member:

```
TAO::Transport_Selection_Guard transport_;
```

This guard automatically keeps track of the currenty selected Transport from within its constructor and destructor. The rest of the TC framework makes sure this pointer is stored in a thread-specific storage, by adding an additional member to TSS_Resources:

```
TAO::Transport_Selection_Guard* tsg_;
```

The idea is to keep a pointer to the last guard on the current thread. Each guard keeps a pointer to the previous, effectively creating a stack of transport selection guards. The stack structure ensures both that the selection/deselection of a Transport will be correctly handled. It also ensures that, in case the current thread temporarily changes the Transport, the previous “current” transport will be preserved, no matter how many times such change occurs. A good example for this is a nested up-call scenario.

Inside an interceptor, one can use the methods from Transport Current to obtain information on the currently selected transport. The implementation simply looks up the TAO_Transport pointer via TSS_Resources::tsg_ and obtains the requested data.

10.7. Server Side: Request Processing

On the server side, the TAO_ServerRequest instance already has a Transport pointer. The TAO_ServerRequest lives on the stack, starting its life inside a call to TAO_GIOP_Message_Base::process_request().

Similarly to the client-side, we changed the TAO_ServerRequest to add a field:

```
TAO::Transport_Selection_Guard transport_;
```

Operation is similar to the client-side case. In the collocated case there may not be a transport available, so the TSS slot will be null.

Inside an interceptor then, one can use an RIR-resolved TransportCurrent to create a specialization of TransportInfo, based on the kind of Transport used. Then they would _downcast() it to the specific type.

10.8. Structural and Footprint Impact

As the IIOP implementation of the Transport Current functionality requires additional data to be kept about the Transport, we added a new field to TAO_Transport:

```
/// Transport statistics
TAO::Transport::Stats* stats_
```

TAO::Transport::Stats is a simple class, which keeps track of useful statistical information about how a transport is used:

```
class TAO_Export Stats
{
public:
  Stats ();

  void messages_sent (size_t message_length);
  CORBA::LongLong messages_sent (void) const;
  CORBA::LongLong bytes_sent (void) const;

  void messages_received (size_t message_length);
  CORBA::LongLong messages_received (void) const;
  CORBA::LongLong bytes_received (void) const;

  void opened_since (const ACE_Time_Value&amp; tv);
  const ACE_Time_Value&amp; opened_since (void) const;

private:
  CORBA::LongLong messages_rcvd_; // 32bits not enough (?)
  CORBA::LongLong messages_sent_; // 32bits not enough (?)

  ACE_Basic_Stats bytes_rcvd_;
  ACE_Basic_Stats bytes_sent_;

  ACE_Time_Value  opened_since_;
};
```

To gather the statistics the TAO_Transport::send_message_shared() and TAO_Transport::process_parsed_messages() must be modified. These are non-virtual methods and are being called as part of request and reply processing regardless of what the most derived Transport type is. This property ensures that any specific Transport will have access to these statistics.

10.9. Performance Impact

As the implementation of the Transport Current functionality necessitates some additional processing on the critical path of an invocation, we are expecting a performance impact when the functionality is being used.

It is possible at build time, to disable the functionality, so that applications only incur the penalty if they require the features. The ORB, by default enables the Transport::Current functionality. Adding `transport_current=0` to your default.features file will disable it.

10.10. Example Code

Look at `$TAO_ROOT/tests/TransportCurrent` for code which illustrates and tests this feature.

Chapter 11. Security

11.1. Using SSLIOP

11.1.1. Loading and Configuring the SSLIOP Pluggable Protocol

TAO implements SSL as a pluggable protocol. As such, it must be dynamically loaded into the ORB. You must use a service configurator file to do this. In this case you have to create a `svc.conf` file that includes:

```
dynamic SSLIOP_Factory Service_Object *
  TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() ""
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Note that `TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory` is part of the first line. This will load the SSLIOP protocol from the library called `TAO_SSL` and then use that protocol in the ORB.

11.2. SSLIOP Options

Once the SSLIOP protocol is loaded you may want to setup the private key and certificate files, the authentication level and similar features. This is done by setting more options in the service configurator file, for example:

```
dynamic SSLIOP_Factory Service_Object *
  TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory()"-SSLAuthenticate SERVER"
```

will enforce validation of the server certificate on each SSL connection. The complete list of options is in the following table.

Table 10. SSLIOP Options

Option	Description
`-SSLNoProtection`	On the client side, this option forces request invocations to use the standard insecure IIOP protocol. On the server side, use of this option allows invocations on the server to be made through the standard insecure IIOP protocol. Request invocations through SSL may still be made. This option will be deprecated once the `SecurityLevel2::SecurityManager` interface as defined in the CORBA Security Service is implemented.

Option	Description
`-SSLCertificate FORMAT:filename`	Set the name of the file that contains the certificate for this process. The file can be in Privacy Enhanced Mail (`PEM`) format or ASN.1 (`ASN1`). Remember that the certificate must be signed by a Certificate Authority recognized by the client.
`-SSLPrivateKey FORMAT:filename1`	Set the name of the file that contains the private key for this process. The private key and certificate files must match. It is extremely important that you secure your private key! By default the `OpenSSL` utilities will generate pass phrase protected private key files. The password is prompted when you run the CORBA application.
`-SSLAuthenticate which`	Control the level of authentication. The argument can be `NONE`, `SERVER`, `CLIENT` or `SERVER_AND_CLIENT`. Due to limitations in the SSL protocol `CLIENT` implies that the server is authenticated too.
`-SSLAcceptTimeout which`	Set the maximum amount of time to allow for establishing a SSL/TLS passive connection, i.e. for accepting a SSL/TLS connection. The default value is `10` seconds. See the discussion in 1348 for the rationale behind this option.
`-SSLDHParams filename1`	Set the filename containing the Diffie-Hellman parameters to be used when using DSS-based certificates. The specified file may be a file containing only Diffie-Hellman parameters created by “openssl dhparam”, or it can be a certificate containing a PEM encoded set of Diffie-Hellman parameters.

11.3. Environment variables

The SSLIOP protocol supports the environment variables listed in this table to control its behavior.

Table 11. SSLIOP Environment Variables

Environment Variable	Description
`SSL_CERT_FILE filename`	The name of the file that contains all the trusted certificate authority self-signed certificates. By default it is set to the value of the `ACE_DEFAULT_SSL_CERT_FILE` macro.

Environment Variable	Description
`SSL_CERT_DIR directory`	The name of the directory that contains all the trusted certificate authority self-signed certificates. By default it is set to the value of the `ACE_DEFAULT_SSL_CERT_DIR` macro. This directory must be indexed using the OpenSSL format, i.e. each certificate is aliased with the following link: [source] ---- $ ln -s cacert.pem `openssl x509 -noout -hash $lt; cacert.pem`.0 ---- Consult the documentation of your SSL implementation for more details.
`SSL_EGD_FILE filename`	The name of the UNIX domain socket that the Entropy Gathering Daemon (EGD) is listening on.
`SSL_RAND_FILE filename`	The file that contains previously saved state from OpenSSL's pseudo-random number generator.

11.4. Using the SSLIOP::Current Object

TAO's SSLIOP pluggable protocol allows an application to gain access to the SSL session state for the current request. For example, it allows an application to obtain the SSL peer certificate chain associated with the current request so that the application can decide whether or not to reject the request. This is achieved by invoking certain operations on the `SSLIOP::Current` object. The interface for `SSLIOP::Current` object is:

```
module SSLIOP
{
# pragma prefix "omg.org"

  /// A DER encoded X.509 certificate.
  typedef sequence<octet> ASN_1_Cert;

  /// A chain of DER encoded X.509 certificates. The chain
  /// is actually a sequence. The sender's certificate is
  /// first, followed by any Certificate Authority
  /// certificates proceeding sequentially upward.
  typedef sequence<ASN_1_Cert> SSL_Cert;

  /// The following are TAO extensions.
# pragma prefix "ssliop.tao"

  /// The SSLIOP::Current interface provides methods to
  /// gain access to the SSL session state for the current
  /// execution context.
  local interface Current : CORBA::Current
  {
    /// Exception that indicates a SSLIOP::Current
    /// operation was invoked outside of an SSL
    /// session.
    exception NoContext {};

    /// Return the certificate chain associated with
    /// the current execution context. If no SSL
    /// session is being used for the request or
    /// upcall, then the NoContext exception is
    /// raised.
    SSL_Cert get_peer_certificate_chain ()
      raises (NoContext);
  };

# pragma prefix "omg.org"
};
```

Obtaining a Reference to the `SSLIOP::Current` Object

A reference to the `SSLIOP::Current` object may be obtained using the standard `CORBA::ORB::resolve_initial_references()` mechanism with the argument `"SSLIOPCurrent"`. Here is an example:

```
int argc = 0;
CORBA::ORB_var orb = CORBA::ORB_init (argc, "", "my_orb");
CORBA::Object_var obj =
  orb->resolve_initial_references ("SSLIOPCurrent");
SSLIOP::Current_var ssliop =
  SSLIOP::Current::_narrow (obj.in ());
```

Examining the Peer Certificate for the Current Request Using OpenSSL Once a reference to the `SSLIOP::Current` object has been retrieved, the peer certificate for the current request may be obtained by invoking the `SSLIOP::get_peer_certificate` method, as follows:

```
// This method can throw a SSLIOP::Current::NoContext
// exception if it is not invoked during a request being
// performed over SSL.
SSLIOP::ASN_1_Cert_var cert =
  ssliop->get_peer_certificate ();
```

The retrieved X.509 peer certificate is in DER (a variant of ASN.1) format. DER is the on-the-wire format used to transmit certificates between peers.

OpenSSL can be used to examine the certificate. For example, to extract and display the certificate issuer from the DER encoded X.509 certificate, the following can be done:

```
#include <openssl/x509.h>
#include <iostream>

// Obtain the underlying buffer from the
// SSLIOP::ASN_1_Cert.
CORBA::Octet *der_cert = cert->get_buffer ();
char buf[BUFSIZ];

// Convert the DER encoded X.509 certificate into
// OpenSSL's internal format.
X509 *peer = ::d2i_X509 (0, &amp;der_cert, cert->length ());
::X509_NAME_oneline (
      ::X509_get_issuer_name (peer),
      buf,
      BUFSIZ);

std::cout "Certificate issuer:" << buf << std::endl;

::X509_free (peer);
```

Chapter 12. Real Time CORBA

The RTCORBA specification contains a lot of different features. This chapter will give an overview of all the features and how to use them in the extended version of this guide

12.1. Protocol Policies

The Real Time CORBA specification contains a part describing the protocol properties. There are some discussions within the OMG to move these protocol properties to the core spec. In addition to `TCPProtocolProperties` defined by the Real-Time CORBA specification, TAO provides configurable properties for each protocol it supports. With these properties you can tune the underlying protocol for your application requirements. Below is a summary of all protocol properties available in TAO. For each protocol we list the Profile Id, whether it is TAO specific, the IDL interface, the class it implements and the method on the `RTORB` which you can use to create an instance of the properties.

12.1.1. IIOP

- Protocol Profile Id: 0
- TAO specific: no
- IDL Interface: RTCORBA::TCPProtocolProperties
- Implementation class: TAO_TCP_Properties
- RTORB method: create_tcp_protocol_properties

Table 12. IIOP Protocol Properties

Attribute	Default value
long send_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
long recv_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
boolean keep_alive	true
boolean dont_route	false
boolean no_delay	true
enable_network_priority	false

12.1.2. UIOP

- Protocol Profile Id: 0x54414f00U
- TAO specific: yes
- IDL Interface: RTCORBA::UnixDomainProtocolProperties
- Implementation class: TAO_UnixDomain_Protocol_Properties
- RTORB method: create_unix_domain_protocol_properties

Table 13. UIOP Protocol Properties

Attribute	Default value
long send_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
long recv_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ

12.1.3. SHMIOP

- Protocol Profile Id: 0x54414f02U
- TAO specific: yes
- IDL Interface: RTCORBA::SharedMemoryProtocolProperties
- Implementation class: TAO_SharedMemory_Protocol_Properties
- RTORB method: create_shared_memory_protocol_properties

Table 14. SHMIOP Protocol Properties

Attribute	Default value
Attribute	Default value
long send_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
long recv_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
boolean keep_alive (not yet supported)	true
boolean dont_route (not yet supported)	false
boolean no_delay	true
long preallocate_buffer_size	not yet supported
string mmap_filename	not yet supported
string mmap_lockname	not yet supported

12.1.4. DIOP

- Protocol Profile Id: 0x54414f04U
- TAO specific: yes
- IDL Interface: RTCORBA::UserDatagramProtocolProperties
- Implementation class: TAO_UserDatagram_Protocol_Properties
- RTORB method: create_user_datagram_protocol_properties

Table 15. DIOP Protocol Properties

Attribute	Default value
long send_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ

Attribute	Default value
long recv_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
enable_network_priority	false

12.1.5. SCIOP

- Protocol ProfileId: 0x54414f0EU
- TAO specific: yes
- IDL Interface: RTCORBA::StreamControlProtocolProperties
- Implementation class: TAO_StreamControl_Protocol_Properties
- RTORB method: create_stream_control_protocol_properties

Table 16. SCIOP Protocol Properties

Attribute	Default value
long send_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
long recv_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
boolean keep_alive (not yet supported)	true
boolean dont_route (not yet supported)	false
boolean no_delay	true
enable_network_priority	false

12.2. Creating the protocol properties

Real-Time CORBA 1.0 does not define how protocol properties are created.
TAO_Protocol_Factory class can be used to create default ProtocolProperties for a particular protocol given its ProfileId:

```
class TAO_Protocol_Properties_Factory
{
public:
  static RTCORBA::ProtocolProperties*
  create_transport_protocol_property (IOP::ProfileId id);

  static RTCORBA::ProtocolProperties*
  create_orb_protocol_property (IOP::ProfileId id);
};
```

The RTORB delivers a set of methods to create the different types of protocol properties. The code fragment below shows how you can use these methods.

```
// Retrieve the RTORB and narrow it to the derived interface
CORBA::Object_var object =
  orb->resolve_initial_references ("RTORB");

RTCORBA::RTORB_var rt_orb =
  RTCORBA::RTORB::_narrow (object.in ());

// Create the protocol properties, replace XXX with the type you want to
create
RTCORBA::XXXProtocolProperties_var protocol_properties =
  rt_orb->create_xxx_protocol_properties (....);

// Add the protocol properties to a list
RTCORBA::ProtocolList protocols;
protocols.length (1);
protocols[0].protocol_type = 0;
protocols[0].transport_protocol_properties =
  RTCORBA::ProtocolProperties::_duplicate (tcp_properties.in ());
protocols[0].orb_protocol_properties =
 RTCORBA::ProtocolProperties::_nil ();

CORBA::PolicyList policy_list;
policy_list.length (1);
policy_list[0] = rt_orb->create_client_protocol_policy (protocols);
```

The protocol properties can be set on different levels. The possible levels are `ORB`, `THREAD`, and `OBJECT` level. The following code fragmens show how to set them at a certain level. First, let us set the policy at ORB level.

```
object = orb->resolve_initial_references ("ORBPolicyManager");

CORBA::PolicyManager_var policy_manager =
  CORBA::PolicyManager::_narrow (object.in ());

policy_manager->set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
```

You can set them at THREAD level using the following code fragment.

```
object = orb->resolve_initial_references ("PolicyCurrent");

CORBA::PolicyCurrent_var policy_current =
  CORBA::PolicyCurrent::_narrow (object.in ());

policy_current->set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
```

And as last you can set the protocol properties at object level.

```
CORBA::Object_var object = server->_set_policy_overrides (policy_list,
CORBA::SET_OVERRIDE);
server = Test::_narrow (object.in ());
```

Alternatively, concrete ProtocolProperties implementation classes can be instantiated directly as needed.

TAO delivers also a non RTCORBA way of setting the send and receive buffer sizes. These can be passed to the `CORBA::ORB_init` call. The options to specify are `-ORBSndSock` and `-ORBRcvSock`. In case you use these options and RTCORBA, the RTCORBA setting do override these options.

Protocol policies do not depend on any other RTCORBA features and can be used alone. In fact, we plan to make protocol policies available outside RTCORBA, and better integrate them with the Pluggable Protocols framework in the near future.

Chapter 13. CORBA/e

13.1. The standard

CORBA/e dramatically minimizes the footprint and overhead of typical middleware, while retaining the core elements of interoperability and real-time computing that support optimized distributed systems. The CORBA/e standard contains two profiles, CORBA/e Compact and CORBA/e Micro Profile. Tailored separately for minimal and single-chip environments, the Compact Profile and the Micro Profile bring industry-standard interoperability and real-time predictable behavior to Distributed Real-time and Embedded (DRE) computing. CORBA/e is available as ptc/2006-08-03.

13.2. CORBA/e Compact Profile

CORBA/e Compact Profile merges key features of standard CORBA suitable for resource-constrained static systems (no DII, DSI, Interface Repository, or Component support) and Real-time CORBA into a powerful yet compact middleware package that interoperates with other CORBA clients and servers of every scale, executes with the deterministic characteristics required of a true real-time platform, and leverages the knowledge and skills of your existing development team through its mature industry-standard architecture.

13.3. CORBA/e Micro Profile

The CORBA/e Micro Profile shrinks the footprint even more, small enough to fit low-powered microprocessors or digital signal processors (DSPs). This profile further eliminates the Valuetype, the Any type, most of the POA options preserved in the Compact Profile, and all of the Real-time functions excepting only the Mutex interface. In exchange for these limitations, the profile defines a CORBA executable that vendors have fit into only tens of kilobytes small enough to fit onto a high-end DSP or microprocessor on a hand-held device.

13.4. TAO support

TAO supports CORBA/e compact and micro but we have not checked all small details of the spec to get out all required functionality. We have updated the source code of TAO to support most global options, but we can reduce the footprint even more. To use CORBA/e compact or micro we advice you to obtain the source only package of TAO (as described in this chapter). Then you can add `corba_e_compact=1` or `corba_e_micro=1` to the `default.features` file and regenerate the makefiles using MPC.

Chapter 14. ACE documentation

When using TAO you automatically also are using ACE. ACE itself is powerful and you can use ACE also together with TAO in your application. This guide is focused on TAO, if you want to know more about ACE we do recommend the following books.

14.1. C++NPv1

C++ Network Programming: Mastering Complexity Using ACE and Patterns (paid link) describes how middleware and the ACE toolkit help address key challenges associated with developing networked applications. We review the core native OS mechanisms available on popular OS platforms and illustrate how C++ and patterns are applied in ACE to encapsulate these mechanisms in class library wrapper facades that improve application portability and robustness. The book's primary application example is a networked logging service that transfers log records from client applications to a logging server. C++NPv1 was published in mid-December, 2001. The Table of Contents is available online.

14.2. C++NPv2

C++ Network Programming: Systematic Reuse with ACE and Frameworks (paid link) describes a family of object-oriented network programming frameworks provided by the ACE toolkit. These frameworks help reduce the cost and improve the quality of networked applications by reifying proven software designs and implementations. ACE's framework-based approach expands reuse technology far beyond what can be achieved by reusing individual classes or even class libraries. We describe the design of these frameworks, show how they can be applied to real networked applications, and summarize the design rules that underlie the effective use of these frameworks. C++NPv2 was published in early November, 2002. The Table of Contents is available online.

14.3. ACE Programmer's Guide

APG (paid link) is a practical, hands-on guide to ACE for C++ programmers building networked applications and next-generation middleware. The book first introduces ACE to beginners. It then explains how you can tap design patterns, frameworks, and ACE to produce effective, easily maintained software systems with less time and effort. The book features discussions of programming aids, interprocess communication (IPC) issues, process and thread management, shared memory, the ACE Service Configurator framework, timer management classes, the ACE Naming Service, and more.

Chapter 15. CORBA Books

This chapter gives a list of other CORBA books that you can use as a reference. It is our advice that you get a copy of all these books on your desk. Each book has its own specific topics and value.

15.1. Advanced CORBA® Programming with C++

Advanced CORBA® Programming with C++ (paid link) is a book you must have when using CORBA. It explains a lot of details and gives a lot of examples. The only concern is that the book is a little bit outdated.

15.2. Pure CORBA

Pure CORBA (paid link) is a useful book that explains some of the newer features. It has example code in C++ and Java.

15.3. CORBA Explained Simply

CORBA Explained Simply (paid link) is a very good starter book which is available for free.

Chapter 16. Design books

This chapter gives a list of other design/architecture books that you can use as a reference. It is our advice that you get a copy of all these books on your desk. Each book has its own specific topics and value.

16.1. POSA2

The POSA2 (paid link) book describes all major ACE design patters.

Chapter 17. C++ books

This chapter gives a list of C++ books that you can use as a reference. It is our advice that you get a copy of all these books on your desk. Each book has its own specific topics and value.

17.1. The C++ Programming Language

This book (paid link) describes the C++ Language.

17.2. Modern C++ Design

This book (paid link) describes the concept of generic components within the C++ language.

Chapter 18. Frequently asked questions

Question	Solution
Can I use ACE/TAO/CIAO on Windows 95/98/ME?	Any version before x.5.6 can be build and used on Windows 95/98/ME. Newer version don't have support for these Windows versions anymore.
Can I use ACE/TAO/CIAO on OpenVMS?	Any version after x.5.3 can be build and used on out of the box on OpenVMS 8.2 Alpha. The Itanium port for OpenVMS 8.3 is also ready. The main sponsor of this port is upgrading their production systems to Itanium and because of that we are ending maintenance for the Alpha port January 2009.
What is the latest version of ACE/TAO/CIAO that is supported with Visual Studio 6?	The latest version that is supported with Visual Studio 6 is x.5.1. Any versions after this release won't build anymore with Visual Studio 6.
What happened with all the C++ environment macros?	TAO has supported for years platforms that lack native C++ exception support. Around TAO 1.4.8 we identified several problems with the support for emulated exceptions. At that moment the macros where deprecated and we didn't maintain them anymore. Because no party was interested in funding the maintenance of the support for platforms lacking native C++ exceptions the macros where fully removed from the TAO source code. With TAO 1.5.6 part of the macros where removed, with TAO 1.5.7 all environment macros are removed.
I am using TAO with SSLIOP but can't retrieve the peer certificate, what do I do wrong?	There is a known bug in TAO 1.5.{2,3,4,5,6} which caused that when you retrieve the peer certificate you get an exception or no data. This bug has been fixed in TAO 1.5.7 and newer.
How do I get a TAO logfile that has timestamps?	From TAO 1.4.1 you can pass `-ORBVerboseLogging 2` to the ORB_init call to add a timestamp to each log line.

Question	Solution
I am using Fedora Core 6, Fedora Core 7, or RedHat Enterprise 5 and I do get unresolved externals on `ACE_Obstack_T`, what can I do?	Fedora Core 6, Fedora Core 7, and RedHat Enterprise 5 get shipped with GCC 4.1.{1,2} which has a fix for a problem we encountered in the past. The workaround for this problem now causes the problem. For ACE/TAO/CIAO x.5.10 and earlier you have to enable a workaround, with x.6 we do automatically detect the problematic GCC versions with FC6, FC7, and RHEL5. You have to enable the workaround using the following in your `config.h` file using `#define ACE_GCC_HAS_TEMPLATE_INSTANTIATION_VISIBILITY_ATTRS 1`
How do I enable or disable the expansion of the ACE_DEBUG macro?	If you want to enable the expansion of the ACE_DEBUG macro use the following in your config.h file and recompile ACE using `#define ACE_NDEBUG 0` to enable it or `#define ACE_NDEBUG 1` to disable it
I can't unpack the distribution on Solaris, what is happening?	The distribution is created with GNU tar, the Solaris tar can't handle this tar file and will fail. Download the GNU tar from Sunfreeware.com and use that tar utility.
What is the latest version of ACE/TAO/CIAO that is maintained for VxWorks 5.5.x?	The latest version is x.6.6. After this micro we did end the daily maintenance due to the lack of funding for this effort. Reinstating this port is technically possible but needs funding.
I am getting unresolved externals when building soreduce on Ubuntu 7.04	This is a known problem in the GNU toolchain of Ubuntu 7.04. This can be resolved by adding `no_hidden_visibility=1` to your `platform_macros.GNU` file. This is not needed anymore if you are using ACE/TAO/CIAO x.6.2 or newer.

Chapter 19. Building TAO

Dependent on your operating system, compiler and your requirements there are different ways how to build TAO. This chapter gives an overview of the different types of build you can perform.

19.1. Microsoft Visual Studio

ACE contains project files for Microsoft Visual Studio 2017 (vc141), and Visual Studio 2019 (vc142). Visual Studio supports building for desktop/server Windows as well as for Windows CE and Windows Mobile. Since not all users will be interested in the CE/Mobile capability, these platforms have separate solution and project files from the desktop/server Windows. Furthermore, vc141 and vc142 use different file formats but the same file suffixes (.sln and .vcproj). To support both environments, ACE supplies files with different names for the different development and target platforms. The platform/name mapping is shown in the following table. All solution files have a .sln suffix and all project files have a .vcproj suffix.

Table 17. MSVC Solutions

Platform	Filename
Visual Studio 2017 for desktop/server	name_vs2017
Visual Studio 2019 for desktop/server	name_vs2019

In order to compile ACE/TAO using Visual Studio use the following steps:

- Decompress the ACE distribution into a directory, where it will create a `ACE_wrappers` directory containing the distribution. The `ACE_wrappers` directory will be referred to as `ACE_ROOT` in the following steps so `ACE_ROOT\ace` would be `C:\ACE_wrappers\ace` if you decompressed into the root directory.
- Add the `ACE_wrappers\lib` location as full path to the `PATH` system environment variable.
- Create a file called config.h in the `ACE_ROOT\ace` directory that contains:

```
#include "ace/config-win32.h"
```

- The static, DLL and MFC library builds are kept in different workspaces. Workspaces DLL builds will be available through the stock release at DOC group's website. The workspaces for static and MFC are not available and have to be generated using MPC. Please see MPC's README for details.
- Now load the solution file for ACE (ACE_ROOT/ACE_vs2019.sln).
- Make sure you are building the configuration (i.e, Debug/Release) the one you'll use (for example, the debug tests need the debug version of ACE, and so on). All these different configurations are provided for your convenience. You can either adopt the scheme to build your applications with different configurations, or use `ace\config.h` to tweak with the default settings on Windows. Note: If you use the dynamic libraries, make sure you include `ACE_ROOT\lib` in your PATH whenever you run programs that uses ACE. Otherwise you

may experience problems finding ace.dll or aced.dll.

- If you want to use the standard C++ headers (iostream, cstdio, etc) that comes with MSVC, then add the line:

```
#define ACE_HAS_STANDARD_CPP_LIBRARY 1
```

before the `#include` statement in `ACE_ROOT\ace\config.h`.

- To use ACE with MFC libraries, also add the following to your config.h file. Notice that if you want to spawn a new thread with CWinThread, make sure you spawn the thread with THR_USE_AFX flag set.

```
#define ACE_HAS_MFC 1
```

By default, all of the ACE projects use the DLL versions of the MSVC runtime libraries. You can still choose use the static (LIB) versions of ACE libraries regardless of runtime libraries. The reason we chose to link only the dynamic runtime library is that almost every Windows system has these library installed and to save disk space. If you prefer to link MFC as a static library into ACE, you can do this by defining `ACE_USES_STATIC_MFC` in your config.h file. However, if you would like to link everything (including the MSVC runtime libraries) statically, you'll need to modify the project files in ACE yourself.

- Static version of ACE libraries are built with `ACE_AS_STATIC_LIBS` defined. This macro should also be used in application projects that link to static ACE libraries

Optionally you can also add the line

```
#define ACE_NO_INLINE
```

before the #include statement in ACE_ROOT\ace\config.h to disable inline function and reduce the size of static libraries (and your executables.)

- ACE DLL and LIB naming scheme:

We use the following rules to name the DLL and LIB files in ACE when using MSVC.

"Library/DLL name" + (Is static library ? "s" : "") + (Is Debugging enable ? "d" : "") + {".dll"|".lib"}

19.2. GNU make

Here's what you need to do to build ACE using GNU Make:

- Install GNU make 3.79.1 or greater on your system. You must use GNU make when using ACE's traditional per-platform configuration method or ACE won't compile.
- Add an environment variable called ACE_ROOT that contains the name of the root of the directory where you keep the ACE wrapper source tree. The ACE recursive Makefile scheme

needs this information. If you build TAO you also need to set TAO_ROOT.

- Create a configuration file, `$ACE_ROOT/ace/config.h`, that includes the appropriate platform and compiler specific header configurations from the ACE source directory. For example:

```
#include "ace/config-linux.h"
```

The platform and compiler-specific configuration file contains the #defines that are used throughout ACE to indicate which features your system supports. If you desire to add some site-specific or build-specific changes, you can add them to your `config.h` file, place them before the inclusion of the platform-specific header file.

There are config files for most versions of UNIX, see this table for an overview of the available config files. If there isn't a version of this file that matches your platform/compiler, you'll need to make one. Please send email to the ace-users list if you get it working so it can be added to the master ACE release.

- Create a build configuration file, `$ACE_ROOT/include/makeinclude/platform_macros.GNU`, that contains the appropriate platform and compiler-specific Makefile configurations, e.g.,

```
include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU
```

This file contains the compiler and Makefile directives that are platform and compiler-specific. If you'd like to add make options, you can add them before including the platform-specific configuration. See this table for an overview of the available platform files. In the following table there is an overview of some of the flags you can specify in your `platform_macros.GNU` file.

NOTE There really is not a # character before 'include' in the platform_macros.GNU file. # is a comment character for GNU make.

- Because ACE builds shared libraries, you'll need to set LD_LIBRARY_PATH (or equivalent for your platform) to the directory where binary version of the ACE library is built into. For example, you probably want to do something like the following:

```
export LD_LIBRARY_PATH=$ACE_ROOT/lib:$LD_LIBRARY_PATH
```

- When all this is done, hopefully all you'll need to do is type:

```
make
```

at the ACE_ROOT directory. This will build the ACE library, tests, the examples, and the sample applications. Building the entire ACE release can take a long time and consume lots of disk space, however. Therefore, you might consider cd'ing into the $ACE_ROOT/ace directory and running

make there to build just the ACE library. As a sanity check, you might also want to build and run the automated one-button tests in $ACE_ROOT/tests. Finally, if you're also planning on building TAO, you should build the gperf perfect hash function generator application in $ACE_ROOT/apps/gperf/src.

Table 18. Available config files

File	Description
config-aix-5.x.h	AIX 5
config-cygwin32.h	Cygwin
config-hpux-11.00.h	HPUX 11i v1/v2/v3
config-linux.h	All linux versions
config-openvms.h	OpenVMS 8.2 and 8.3 on Alpha and IA64
config-netbsd.h	NetBSD
config-sunos5.8.h	Solaris 8
config-sunos5.9.h	Solaris 9
config-sunos5.10.h	Solaris 10
config-vxworks5.x.h	VxWorks 5.5.{1,2}
config-vxworks6.2.h	VxWorks 6.2
config-vxworks6.3.h	VxWorks 6.3
config-vxworks6.4.h	VxWorks 6.4
config-vxworks6.5.h	VxWorks 6.5
config-vxworks6.6.h	VxWorks 6.6
config-vxworks6.7.h	VxWorks 6.7

Table 19. Available platform files

File	Description
platform_aix_g++.GNU	AIX with the GCC compiler
platform_aix_ibm.GNU	AIX with the IBM compiler
platform_cygwin32.GNU	Cygwin
platform_hpux_aCC.GNU	HPUX with the HP aCC compiler
platform_hpux_gcc.GNU	HPUX with the GCC compiler
platform_linux.GNU	All linux versions with the GCC compiler
platform_linux_icc.GNU	All linux versions with the Intel C++ compiler
platform_openvms.GNU	OpenVMS with the HP compiler
platform_sunos5_g++.GNU	Solaris with the GCC compiler

File	Description
platform_sunos5_sunc++.GNU	Solaris with the Sun compiler
platform_vxworks5.5.x.GNU	VxWorks 5.5.{1,2} with the GCC compiler
platform_vxworks6.2.GNU	VxWorks 6.2 with the GCC compiler
platform_vxworks6.3.GNU	VxWorks 6.3 with the GCC compiler
platform_vxworks6.4.GNU	VxWorks 6.4 with the GCC and Diab compiler
platform_vxworks6.5.GNU	VxWorks 6.5 with the GCC and Diab compiler
platform_vxworks6.6.GNU	VxWorks 6.6 with the GCC and Diab compiler
platform_vxworks6.7.GNU	VxWorks 6.7 with the GCC and Diab compiler

Table 20. GNU make options

Opton	Description
buildbits={32 or 64}	Build 32 or 64bit
inline={0 or 1}	Inlining disable or enabled
debug={0 or 1}	Debugging disable or enabled
optimize={0 or 1}	Optimizations disable or enabled
exceptions={0 or 1}	C++ exceptions disable or enabled
static_libs_only=1	Only build static libraries
xt=1	Build with Xt (X11 Toolkit) support
fl=1	Build with FlTk (Fast Light Toolkit) support
tk=1	Build with Tk (Tcl/Tk) support
qt=1	Build with Qt (Trolltech Qt) support
ssl=1	Build with OpenSSL support
rapi=1	Build with RAPI
stlport=1	Build with STLPort support
rwho=1	Build with rwho, this results in building apps/drwho
wfmo=1	Build with wfmo support (Win32 only)
winregistry=1	Build with windows registry support (Win32 only)
winnt=1	Build WinNT-specific projects (Win32 only)

19.3. Embarcadero C++ Builder

Before building ACE/TAO you should check your C++ Builder installation to see if it is supported. An overview of the supported Embarcadero C++ compilers is available at the Remedy IT website.

If your C++ Builder compiler is not on the supported version list it will probably not be possible to build ACE/TAO

19.3.1. Building the libraries

Follow the steps below to build the libraries with C++ Builder.

- Decompress the ACE distribution into a directory, where it will create an ACE_wrappers directory containing the source. The ACE_wrappers directory will be referred to as ACE_ROOT in the following steps so `ACE_ROOT\ace` would be `C:\ACE_wrappers\ace` when you decompressed into the root directory.
- Create a file called config.h in the ACE_ROOT\ace directory that contains:

```
#include "ace/config-win32.h"
```

- Open a Command Prompt (DOS Box).
- Set the ACE_ROOT environment variable to point to the ACE_wrappers directory. For example:

```
set ACE_ROOT=C:\ACE_wrappers
```

- Change to the `ACE_ROOT\ace` directory.
- Build release DLLs for ACE by invoking:

```
make -f Makefile.bmak all
```

The make should be the Embarcadero make, not the GNU make utility. You can set additional environment variables to build a different version of ACE, see the following table for an overview of the supported environment variables.

- Optionally install the ACE header files, libraries and executables for use in your applications. Here we are installing them into C:\ACETAO:

```
make -f Makefile.bor -DINSTALL_DIR=C:\ACETAO install
```

Note that when you run make in a sub directory you give `make -f Makefile.bor all`. The `all` is needed to make sure the complete project is build.

Table 21. Embarcadero Environment Variables

Variable	Description
DEBUG=1	Build a debug version
RELEASE=1	Build a release version

Variable	Description
STATIC=1	Build a static version
UNICODE=1	Build an unicode version
CODEGUARD=1	Build a version with Codeguard support. Should only be used when DEBUG is also set
CPU_FLAG=-6	Build a version optimized for a certain CPU. For this there are special compiler flags (-3/-4/-5/-6), see the Embarcadero help for more info.

19.3.2. Building the ACE regression tests

Before you can build the tests you need to build the protocols directory. Change the directory to ACE_ROOT\protocols and start the build with:

```
make -f Makefile.bmak all
```

The tests are located in ACE_ROOT\tests, change to this directory. You build then the tests with the following command:

```
make -f Makefile.bmak all
```

Once you build all the tests, you can run the automated test script using:

```
perl run_test.pl
```

in the tests directory to try all the tests. You need to make sure the ACE bin and lib directory (in this case `ACE_ROOT\bin` and `ACE_ROOT\lib`) are on the path before you try to run the tests.

19.3.3. Using VCL

You can use ACE/TAO in a VCL application but there are some specific requirements set by the VCL environment. You have to make sure the ACE library is initialized before the VCL libraries and cleanup of the ACE library happens after the VCL library. This can be achieved by using the following code for your `WinMain`

```
#pragma package(smart_init)

void ace_init(void)
{
#pragma startup ace_init
  ACE::init();
}

void ace_fini(void)
{
#pragma exit ace_fini
  ACE::fini();
}

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  // ...
}
```

19.4. MinGW

MinGW delivers the GCC compiler for the Windows platform with the ability to use the Win32 API. The first step is install MinGW by following the following steps.

- Download the installer from www.mingw.org
- Install the standard selection with C++ added, don't install the make utility
- Download MSYS from www.mingw.org
- Install MSYS and in the post setup point it to the location of MinGW

Now you have done this, we can build for MinGW. Be aware that MinGW hasn't been maintained for a long time, but for the x.6 release of ACE/TAO we have made sure you can build ACE/TAO out of the box. Extract the distribution and from the MSYS shell go to the ACE_wrappers directory. Now give:

```
export ACE_ROOT=`pwd`
export TAO_ROOT=$ACE_ROOT/TAO
```

Create the file `$ACE_ROOT/ace/config.h` with the contents:

```
#include "ace/config-win32.h"
```

Create the file `$ACE_ROOT/include/makeinclude/platform_macros.GNU`.

```
include $(ACE_ROOT)/include/makeinclude/platform_mingw32.GNU
```

If you want to use the ACE QoS library you need to regenerate the GNU makefiles. Create the file `bin/MakeProjectCreator/config/default.features` with the contents:

```
qos=1
```

Now regenerate the GNU makefiles by using MPC (make sure ACE_ROOT and TAO_ROOT are both set)

```
perl bin/mwc.pl -type gnuace
```

When you only want to build ACE, give `make` in the $ACE_ROOT/ace directory. If you want to build the full package, give `make` in the $ACE_ROOT directory. A known problem is that the performance test $TAO_ROOT/performance-tests/POA/Demux doesn't compile. This is because of a stack overflow in the GCC compiler which has been reported to the MinGW maintainers. When you want to do a MinGW autobuild using a schedule task you need to execute the build with a special setup. First, create a batch file `mingw_build.bat` with the following contents. `mingw.xml` is the autobuild config file that you have made.

```
cd /c/ACE/autobuild
svn up
cd /c/ACE/autobuild/configs/autobuild/remedynl
perl /c/ACE/autobuild/autobuild.pl mingw.xml
```

Then the batch file to schedule is called `mingw.bat` and has the contents:

```
echo "/c/ACE/autobuild/configs/autobuild/remedynl/mingw_build.bat" |
C:\msys\1.0\bin\sh --login -i
```

19.5. Building a host build

Cross compilation is a setup where you build TAO and your application on one architecture and deploy it on a different architecture. An example setup is use an Intel Linux host and a VxWorks PPC target, a different example is using an Intel Linux host and a Intel RTEMS target. For such setups you will need a host build of ACE/TAO to provide at least gperf and tao_idl. This chapter describes the steps you have to take to setup a minimal host build using a Linux based system.

Start by downloading a distribution from http://download.dre.vanderbilt.edu. Extract the ACE+TAO distribution to a new directory (for example `ACE/host`)

Go to the `ACE_wrappers/` directory and give:

```
export ACE_ROOT=`pwd`
export TAO_ROOT=$ACE_ROOT/TAO
```

Create the file `$ACE_ROOT/ace/config.h` with the contents below when you use a Linux host. If

you have a different host system, see this table for the other files you can include.

```
#include "ace/config-linux.h"
```

Create the file `$ACE_ROOT/include/makeinclude/platform_macros.GNU` with the contents below. If you have a different host system, see the following table for the other files you can include.

```
static_libs_only=1
debug=0
inline=0
optimize=0
include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU
```

If you are using TAO we only need to build a subset of ACE/TAO, to do this, we create the file `$ACE_ROOT/TAO_Host.mwc` with the contents below.

```
workspace {
  $(ACE_ROOT)/ace
  $(ACE_ROOT)/apps/gperf/src
  $(TAO_ROOT)/TAO_IDL
}
```

Now we have to run MPC to generate just the project files for one of these workspaces

```
perl bin/mwc.pl TAO_Host.mwc -type gnuace
```

And now we can build the host tree using GNU make. This will then only build this specific subset of libraries and executables.

For the cross build itself we now have to refer to this host build. You can do this by specifying the location of the host build as `HOST_ROOT` in the environment or into the `platform_macros.GNU` file.

19.6. CE GCC

With CE GCC we have to do a cross build. When you want to compile TAO you first need to setup a host build, from this host build we use the gperf and TAO_IDL tools in the cross build. In the host build section we describe the steps to setup a host build, first follow the steps described there to setup a host build.

Now you have done this, we can build for CE GCC. For this we extract another tree for example to `ACE/cegcc`.

First we go to `ACE/cegcc/ACE_wrappers` and give:

```
export ACE_ROOT=`pwd`
export TAO_ROOT=$ACE_ROOT/TAO
```

Create the file `$ACE_ROOT/ace/config.h` with the contents:

```
#define TEST_DIR ACE_TEXT ("/network/temp/ACE_wrappers/tests/")
#define ACE_DEFAULT_TEST_DIR ACE_TEXT ("/network/temp/ACE_wrappers/tests/")
#include "ace/config-win32.h"
```

The TEST_DIR and ACE_DEFAULT_TEST_DIR is used as directory where the tests store their log files in the log directory. This is needed because CE lacks the concept of a current directory.

Create the file `$ACE_ROOT/include/makeinclude/platform_macros.GNU`

```
wince=1
uses_wchar=1
inline=0
include $(ACE_ROOT)/include/makeinclude/platform_cegcc.GNU
```

Create the file `$ACE_ROOT/bin/MakeProjectCreator/config/default.features`

```
qos=1
wince=1
uses_wchar=1
```

You will have to generate the GNU makefiles for CEGCC using

```
perl $ACE_ROOT/bin/mwc.pl -type gnuace
```

When MPC is ready you can compile ACE/TAO using GNU make.

19.7. RTEMS

With RTEMS we have to do a cross build. The first step is to setup a host build, from this host build we use the gperf and TAO_IDL tools in the cross build. In the host build section we describe the steps to setup a host build, first follow the steps described there to setup a host build.

Now you have done this, we can build for RTEMS. For this we extract another tree for example to `ACE/rtems`. First, you need to setup where RTEMS is installed by setting the following environment variable `RTEMS_MAKEFILE_PATH` (see RTEMS documentation for its exact meaning)

Then we go to `ACE/rtems/ACE_wrappers` and give:

```
export ACE_ROOT=`pwd`
export TAO_ROOT=$ACE_ROOT/TAO
```

Create the file `$ACE_ROOT/ace/config.h` with the contents:

```
#include "ace/config-rtems.h"
```

Create the file `$ACE_ROOT/include/makeinclude/platform_macros.GNU`, where `HOST_ROOT` is the location of the host build above (update the location if you have a different directory setup)

```
HOST_ROOT := /home/build/ACE/host/ACE_wrappers
inline=0
include $(ACE_ROOT)/include/makeinclude/platform_rtems.x_g++.GNU
```

For minimal footprint you can use the `ace_for_tao` subsetting feature which means that only the part of the ACE library is build that is required for TAO. To use this feature create the file `bin/MakeProjectCreator/config/default.features` with the contents:

```
ace_for_tao=1
```

And to the `platform_macros.GNU` file above you have to add on the first line:

```
ace_for_tao=1
```

If you have a RTEMS configuration with network support enabled you can build the full distribution without problems, but if you have disabled networking then only a subset of all the code is build daily.

19.7.1. Additional steps when building RTEMS without network

The following steps are additional when you build **WITHOUT** network support. The fact whether you have network support enabled or disabled is automatically detected by the ACE make infrastructure.

To the config.h file add the following defines which disable all custom protocols that are not usable and enable COIOP which has been designed to work without network support.

```
#define TAO_HAS_IIOP 0
#define TAO_HAS_UIOP 0
#define TAO_HAS_DIOP 0
#define TAO_HAS_SHMIOP 0
#define TAO_HAS_COIOP 1
```

Then you have to create the subset with the file ACE_wrappers/rtems.mwc.

```
workspace {
ace
TAO/tao
TAO/tests/COIOP
TAO/orbsvcs/orbsvcs/CosNaming.mpc
TAO/orbsvcs/orbsvcs/CosNaming_Skel.mpc
TAO/orbsvcs/orbsvcs/CosNaming_Serv.mpc
TAO/orbsvcs/tests/COIOP_Naming_Test
TAO/orbsvcs/orbsvcs/Svc_Utils.mpc
}
```

Then regenerate the GNUmakefiles using

```
perl bin/mwc.pl rtems.mwc -type gnuace
```

Then after this you can give a make and only build the parts that needed when building without network support.

19.7.2. Test output to the screen

All ACE tests try to write to a logfile but some systems don't have a disk. With the following lines in the config.h file the output will go to the screen

```
#define ACE_START_TEST(NAME) const ACE_TCHAR *program = NAME; \
  ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("(%P|%t) Starting %s test at %D\n"), NAME))
#define ACE_END_TEST ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("(%P|%t) Ending %s test
t %D\n"), program));
```

19.7.3. Cleaning a build

To clean a build give:

```
make realclean
```

We advice to rebuild everything when you have made a change in ACE or TAO to prevent strange problems because of conflicting libraries.

19.7.4. Using Bochs

We have used Bochs as virtual pc to test the RTEMS port of ACE/TAO (works on Linux and Windows hosts), you can obtain it from the Bochs website.

19.8. From the ACE/TAO main github repository

It is possible as user of ACE/TAO to clone a version from the github repository and have a look at work in progress. An important note is that the doc_group doesn't advice to use the git version for any real projects. It is available to look at changes but there is no support for any code you get directly from the repository.

19.8.1. Using git to clone the main repository

For cloning the github repository use:

```
git clone https://github.com/DOCGroup/ACE_TAO
```

For MPC:

```
git clone https://github.com/DOCGroup/MPC
```

19.8.2. Creating Makefiles/Solutions

In the github repository no GNUmakefiles and project files for building on the various platforms are available. If you build from git you have to generate these makefiles before building ACE/TAO. For this generation you will need perl 5.8 or higher on your system. For windows users we advice Active State Perl.

To build ACE and associated tests, examples, and associated utility libraries with GNUmakefiles, you must generate GNUmakefiles with MPC:

```
$ACE_ROOT/bin/mwc.pl -type gnuace ACE/ACE.mwc
```

On Windows, with Visual Studio 2017, you must generate solution and project files with MPC:

```
$ACE_ROOT/bin/mwc.pl -type vs2017 ACE/ACE.mwc
```

MPC is capable of generating more types of project types, to see a list of possible project types use:

```
$ACE_ROOT/bin/mwc.pl -help
```

When you only want to generate the project files for the core libraries then instead of `ACE/ACE.mwc` use `TAO/TAO_ACE.mwc`

19.9. WindRiver Workbench 2.6

Starting with ACE/TAO x.6.3 it is possible to generate project files for the WindRiver Workbench version 2.6 (VxWorks 6.4). We have validated the MPC support for the ACE lib, TAO libs and the TAO tests. It could be that there are some features that are not generated yet, if you encounter contact us to get a quote for extending the generator to your needs.

With VxWorks we have to do a cross build. The first step is to setup a host build, from this host build we use the gperf and TAO_IDL tools in the cross build. In the host build section we describe the steps to setup a host build, first follow the steps described there to setup a host build.

The Workbench is using eclipse as framework and then has several WindRiver specific extensions. Part of the generation done by MPC is then specifically for the WindRiver tools, part is for the eclipse environment. The Workbench uses the fixed project filenames `.project`, `.wrproject`, and `.wrmakefile`. In the `.project` file the files in the project are listed, in the `.wrproject` the compiler and linker flags are defined, and in the `.wrmakefile` the custom build rules are defined, like triggering the IDL compiler. By default the files are generated in the same directory as the MPC file. When you have multiple projects in one directory you have to add `-make_coexistence` to the invocation of `mwc.pl` Then for each project a new subdirectory will be created to store the files the workbench needs. If you run `mwc.pl -make_coexistence` from the ACE_wrappers directory you will get a lot of subdirectories in your tree.

By default we generate for the flexible build support, when you want to use standard build use `-value_template standard_build=1`.

To get a project with all dependencies create a new workspace file, f.e. vxworks.mwc

```
workspace vxworks {
  ace
  TAO/tao
  TAO/tests/Hello
}
```

You should generate the project files from a VxWorks development shell or should have executed the wrenv script. With x.6.4 or newer you do execute:

```
set ACE_ROOT=your_path
cd %ACE_ROOT%
perl %ACE_ROOT%\bin\mwc.pl vxworks.mwc -type wb26 -make_coexistence
```

After you have generated the project files you have to import them into your current Workbench workspace with the following steps

- Open the workbench
- Select File, Import, General, Existing Projects into Workspace
- Select your ACE_ROOT directory and let the Workbench search for projects

- Select now the projects you want to import in the Projects list and select Finish
- After importing close the workbench
- Copy the prefs file to its location, see below
- Start the workbench again

The build order of the projects is stored in an eclipse file that is generated as workspace by the `wb26` generator. After you have imported the projects into the Workbench close it and then copy the generated `org.eclipse.core.resources.prefs` file to the `.metadata\.plugins\org.eclipse.core.runtime\.settings` directory of the Workbench and then restart the workbench again. Do note that the build order can only be generated for the projects that are listed in the MPC workspace. The other option is to use subprojects to which you can enable with `-value_template enable_subprojects=1`. There is a bug in Workbench 2.6/3.0 related to the build order, it is ignored if you use `wrws_update` to build the workspace from the commandline. When compiling TAO you need to have tao_idl and gperf available. You can copy tao_idl and gperf manually to the ACE_wrappers\bin directory of your target build or you can specify an alternative tao_idl when generating the workspace like below.

```
perl %ACE_ROOT%\bin\mwc.pl vxworks.mwc -type wb26 -value_template
tao_idl=$(HOST_TAO_IDL)
perl %ACE_ROOT%\bin\mwc.pl vxworks.mwc -type wb26 -value_template
tao_idl=c:\tmp\tao_idl
```

When using the `-expand_vars` by default only the environment variables which match the wildcard `*_ROOT` are expanded. If you want to get other environment variables expanded (like `WIND_BASE`) you can specify these through the `-relative` argument or use a file that you specify with `-relative_file`. For example you can use the following relative_file which expands the environment variables listed.

```
*_ROOT
TAO_ROOT,  $ACE_ROOT/TAO
*_BASE
```

We do have some limitations at this moment because of restrictions in MPC or the Workbench. We are working on resolving the MPC restrictions, the Workbench restrictions have been reported to WindRiver and are already converted to enhancement requests. It is important to get these enhancement requests implemented by WindRiver. As user you can have impact on the importance of these enhancement requests, create a new TSR for WindRiver and ask WindRiver to implement these enhancement requests. Pleas let us know that you have done this so that we can inform our local WindRiver contacts. We also have a large list of POSIX enhancement requests, if you are interested in more POSIX compliance contact us to get this list.

- You need to close the workbench when generating the project files. The WB doesn't detect that the .project/.wrproject/org.eclipse.core.resources.prefs files got changed on disk (WIND00116553)

- You need to import, close, and then change the build order file (WIND00116553)
- When using includes/recursive_includes do not use . as path, but an environment variable which can be expanded to a full path (combined WB/MPC restriction)
- We need to generate full paths in the .project file because WB doesn't support relative files like ../MyHeader.h (WIND00116641)
- There is no dependency between the IDL file and the resulting *C.{h,cpp,inl} and *S.{h,cpp,inl} files. This is because the IDL compiler can't be integrated a real build tool because a custom clean step can't be defined (WIND00117037)

Chapter 20. Autobuild framework

For ACE/TAO we have created an autobuild framework that can be used to build ACE/TAO and run all regression tests automatically. When the build is ready the text output is converted to html with an index page so that you can quickly see if there are errors and warnings.

The autobuild framework requires that you have the following tools preinstalled on your system.

- Perl (ActiveState Perl for Windows hosts)
- Git client
- Cygwin for Windows hosts (install the default packages including rsync/scp)

The first step is to checkout the autobuild repository from subversion. The normal setup is to create an ACE directory, below that directory then autobuild and at the same level all builds are located.

```
mkdir ACE
cd ACE
git clone https://github.com/DOCGroup/autobuild
```

Now you have the autobuild tools on your system you have to setup a new autobuild specifically to your system. The first step would be to determine the type of build you want to setup. Important information to determine to start with are:

- `ace/config.h` contents
- `include/makeinclude/platform_macros.GNU` contents
- `bin/MakeProjectCreator/config/default.features` contents * These files are created in the autobuild setup file.

The autobuild file for a specific build is written in xml. In the autobuild file you can specify several tags.

The first line is the opening:

```
<autobuild>
```

Then we need to specify the configuration of the build. These are grouped together

```
    <configuration>
```

This are environment variables that will be set by the autobuild framework before starting the build process. An environment value is specified using its name and you give it a value. This will override any setting done on the system already. If you want to have your values prefixed to the already set environment setting on the OS, specify `type="prefix"` after the value. As example we assume that you put the build under `/build/ACE`

Several variables must be set including ACE_ROOT and TAO_ROOT. On Unix hosts you must set LD_LIBRARY_PATH, on Windows systems make sure you use the PATH variable.

```
<environment name="ACE_ROOT" value="/build/ACE/ACE_wrappers" />
<environment name="TAO_ROOT" value="/build/ACE/ACE_wrappers/TAO"/>
<environment name="LD_LIBRARY_PATH" value="/build/ACE/ACE_wrappers/lib" />
```

The location where the build is stored on the local disk

```
<variable name="root" value="/build/ACE/" />
```

The name of the logfile that is used during the build

```
<variable name="log_file" value="build.txt" />
```

The location where the logfile will be located

```
<variable name="log_root" value="/build/ACE/" />
```

This configs variable is used by the test framework

```
<variable name="configs" value="Linux Exceptions" />
```

After the environment and variable settings you specify the steps the build performs. First, we have to prevent that we run the same build multiple times, to prevent this we use a socket port on the system, the number can be chosen by the user. If you have multiple builds on your system then giving all builds the same number will make sure only one build runs at the same moment. In the past this was also achieved by creating and checking for a `.disable` but the disadvantage of this approach is that in case of a system restart a manual cleanup has to be performed.

```
<command name="process_listener"  options="localhost:32003" />
```

Then you have to get the sourcecode from the archive:

```
<command name="svn" options="co
svn://svn.dre.vanderbilt.edu/DOC/Middleware/sets-anon/ACE+TAO+CIAO ." />
```

Then you have to make sure that the config.h, platform_macros.GNU, and default.features are created. The contents below are an example which you must change to match your requirements.

```
<command name="file_manipulation"
  options="type=create file=ACE_wrappers/ace/config.h output='#include
\x22ace/config-linux.h\x22\n'"/>
<command name="file_manipulation"
  options="type=create
file=ACE_wrappers/include/makeinclude/platform_macros.GNU output='
    include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU\n'" />
<command name="file_manipulation"
  options="type=create
file=ACE_wrappers/bin/MakeProjectCreator/config/default.features output=''"
/>
```

To give other people insight in what for build you are running you can print several sets of information to the config page. The more info the better and by default we deliver the following commands to print this info. Print the host and os information of the system the build runs on

```
<command name="print_os_version" />
```

Print the version information of the compiler. We do support several compilers, see autobuild/command/check_compiler.pm for the list of compilers that can be checked.

```
<command name="check_compiler" options="gcc" />
```

Print the first line of all ChangeLog files together with the config.h/platform_macros.GNU and default.features file. It is important that you have created these files before you use this command to give the correct information

```
<command name="print_ace_config" />
```

If you are using GNU make this command can be used to print the GNU make information to the config page

```
<command name="print_make_version" />
```

Print the perl version

```
<command name="print_perl_version" />
```

If you are hosting an autoconf build this command prints the version of all required tools

```
<command name="print_autotools_version" />
```

If you are using valgrind to detect memory leaks this prints the valgrind version

```
<command name="print_valgrind_version" />
```

With these commands in you autobuild file other users can see the settings of the build and on what for system it runs, this can help when your build has compile and/or test failures.

Generate the makefiles

```
<command name="generate_workspace" options="-exclude TAO/TAO_*.mwc -type
gnuace" />
```

Run make

```
<command name="make" options="-k" />
```

Stop logging

```
<command name="log" options="off" />
```

Remove old logs and convert the txt log file to html

```
<command name="process_logs" options="clean move prettify index" />
```

By default 10 build logs are store, if you want to have less or more build logs, you can specify this as part of the clean step, for example `clean=20` will result in 20 build logs being kept as history. At the moment the specified maximum has been reached the oldest log file will be removed.

Besides the variables mentioned above the following variables can be useful. For example the program used as make defaults to `make` but you can override this for example to `gmake`.

```
<variable name="make_program" value="gmake"/>
```

The program used as subversion client (svn)

```
<variable name="svn_program" value="/usr/local/bin/svn" />
```

The program that will be used for rsync ssh.

```
<environment name="RSYNC_RSH" value="ssh" />
```

The doc_group welcomes daily builds that are hosted by users. To be able to use the build results you will need to commit yourself to run the build at least once a day and publish the build results on a public webserver. To list the build on the scoreboard please change the template below for

your build and send it to one of the mailing lists including the info on which scoreboard this should be listed (ACE, TAO, or CIAO). The website should list at least an email address where we can contact you in case of issues.

```
<build>
<name>BuildName</name>
<url>http://yourserver/yourlogfiledirectory</url>
<build_sponsor>YourComapny</build_sponsor>
<build_sponsor_url>YourWebsite</build_sponsor_url>
</build>
```