

TAO Programmers Guide

Remedy IT

version: 0.11



Table of Contents

Abbreviations	5
1 Preface	6
2 Contact information	7
3 Remedy IT Support conditions for 2007	8
4 Training	9
4.1 Using the ACE C++ Framework	9
4.2 Introduction to CORBA	11
4.3 CORBA Programming with C++	12
4.4 Advanced CORBA Programming with TAO (for Real-Time)	14
5 How to obtain ACE/TAO/CIAO	16
5.1 Major release	16
5.2 Minor release	16
5.3 Bug fix only release	16
5.4 Beta release	17
6 Getting started, hello world	18
6.1 Define your application IDL	18
6.2 Implement the server	19
6.3 Implement the client	22

6.4	Compile client and server	24
6.5	Run your application	26
7	IDL Compiler	27
7.1	Generated Files	27
7.2	Environment Variables	27
7.3	Operation Demuxing Strategies	28
7.4	Collocation Strategies	29
7.5	Output File options	30
7.6	Controlling code generation	30
7.7	Backend options	33
7.8	Other options	35
8	TAO libraries	36
9	Compression	41
9.1	Using compression	41
9.2	Implementing your own compressor	43
10	Using the TAO::Transport::Current Feature	47
10.1	Scope and Context	47
10.2	Programmer's Reference	47
10.3	User's Guide	50
10.4	Configuration, Bootstrap, Initialization and Operation	51
10.5	Implementation and Required Changes	52
10.6	Structural and Footprint Impact	53
10.7	Performance Impact	54
10.8	Example Code	55

11	Security	56
11.1	Using SSLIOP	56
11.2	SSLIOP Options	56
11.3	Environment variables	57
11.4	Using the SSLIOP::Current Object	58
12	Real Time CORBA	61
12.1	Protocol Policies	61
13	CORBA/e	67
13.1	The standard	67
13.2	TAO support	68
14	ACE documentation	69
14.1	C++ Network Programming: Mastering Complexity Using ACE and Patterns	69
14.2	C++ Network Programming: Systematic Reuse with ACE and Frameworks	69
14.3	ACE Programmer's Guide	69
15	CORBA Books	71
16	Frequently asked questions	72
17	Building ACE/TAO/CIAO	74
17.1	Building with Microsoft Visual C++	74
17.2	Building with GNU make	76
17.3	Building with C++ Builder	82
17.4	Building a host build	85
17.5	Building for RTEMS	86
17.6	Building from the subversion repository	89

18	Autobuild framework	92
19	ACE/TAO/CIAO changes	97
19.1	Changes in 5.5.9/1.5.9/0.5.9	97
19.2	Changes in 5.5.8/1.5.8/0.5.8	98
19.3	Changes in 5.5.7/1.5.7/0.5.7	100
19.4	Changes in 5.5.6/1.5.6/0.5.6	103
19.5	Changes in 5.5.5/1.5.5/0.5.5	103
19.6	Changes in 5.5.4/1.5.4/0.5.4	107
19.7	Changes in 5.5.3/1.5.3/0.5.3	109
19.8	Changes in 5.5.2/1.5.2/0.5.2	112
19.9	Changes in 5.5.1/1.5.1/0.5.1	113
19.10	Changes in 5.5.4/1.5.4/0.5.4	114
19.11	Changes in 5.4.10/1.4.10/0.4.10	114
19.12	Changes in 5.4.9/1.4.9/0.4.9	115
19.13	Changes in 5.4.8/1.4.8/0.4.8	115
19.14	Changes in 5.4.7/1.4.7/0.4.7	116
19.15	Changes in 5.4.6/1.4.6/0.4.6	116
19.16	Changes in 5.4.5/1.4.5/0.4.5	117
19.17	Changes in 5.4.4/1.4.4/0.4.4	117
19.18	Changes in 5.4.3/1.4.3/0.4.3	118
19.19	Changes in 5.4.2/1.4.2/0.4.2	118
19.20	Changes in 5.4.1/1.4.1/0.4.1	119
19.21	Changes in 5.4.0/1.4.0/0.4.0	119
20	TPG Changes	120
20.1	0.11	120
20.2	0.10	120

Abbreviations

IOP	InterORB Protocol
ORB	Object Request Broker
TAO	The ACE ORB
TPG	TAO Programmers Guide
ZIOP	Zipped IOP

Preface 1

This is the TAO Programmers Guide (TPG) published by [Remedy IT](#). This first release is published for free.

Based on the user feedback and requests we will extend the TPG on a regular basis. This extended TPG will be available for a small fee (copy based and subscription based). A minimal version of the TPG will always be available for free.

Contact information

2

Our mailing address:

Remedy IT
Postbus 101
2650 AC Berkel en Rodenrijs
The Netherlands
Telephone: +31(0)10 5220139
Fax: +31(0)33 2466511

Our visiting address:

Koperslager 8b
3861 SJ Nijkerk
The Netherlands

For inquiries related to our ACE/TAO/CIAO services and training you can contact us at theaceorb@remedy.nl

If you have a support contract with Remedy IT you can report issues regarding ACE to acesupport@remedy.nl, for issues regarding TAO use taosupport@remedy.nl, and for CIAO use ciaosupport@remedy.nl. We prefer that you use these addresses instead of contacting our staff directly.

If you have remarks regarding the TAO Programmers Guide you can send your remarks to tpg@remedy.nl.

Our general website is online at www.remedy.nl, the section dedicated to ACE/TAO/CIAO can be found on www.theaceorb.nl.

The integrated build and test scoreboard we provide for the developers and user community of ACE/TAO/CIAO can be found online at scoreboard.theaceorb.nl.

Remedy IT Support conditions for 2007

3

In order to obtain support from Remedy IT it is necessary to sign a supportcontract which you can download from our [website](#). Our standard support is based on best effort response and only the hours used for the issues you report will be invoiced. Based on regular reports you are at all times authorized to stop the effort.

If you want to have more guarantees about response time you must sign a support contract with guaranteed hours in blocks of 8 hours. In this case the issues are handled with higher priority. The exact amount of guaranteed hours depends on your requirements regarding response time. Invoice will take place at the end of each month with a minimum of the guaranteed hours.

For both supportcontracts the price/hour will be €70.00, excl. Dutch VAT. The price will be subject to yearly modification and payment within 30 days.

Call-out charges for on-site support within The Netherlands:

- €25,00 with a distance shorter than 50 km
- €50,00 with a distance longer than 50 km

Call-out charges for on-site support outside The Netherlands will be agreed upon beforehand and consist of travel and subsistence expenses.

The support contracts with a guaranteed number of hours are valid for one year. Contact us for special offers on support contracts for other contract periods.

For support questions please use the problem report form that you can find in the ACE/TAO/CIAO package in the ACE_wrappers directory. After having signed a support contract please send your issues to:

- ACE: acesupport@remedy.nl
- TAO: taosupport@remedy.nl
- CIAO: ciaosupport@remedy.nl

Training 4

We provide several courses as part of our training program. These courses are organized as open enrollment at our own course location but also as onsite training at a location of your choice. At this moment we provide the following courses.

- Using the ACE C++ Framework
- Introduction to CORBA
- CORBA Programming with C++
- Advanced CORBA Programming with TAO

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

4.1 Using the ACE C++ Framework

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

Audience

- Software developers moving to distributed applications using ACE

Duration

- 4 days

Prerequisites

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

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

Format

- Lecture and programming exercises

Material

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

Schedule

- The schedule of this course can be found on our [website](#)

4.2 Introduction to CORBA

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 future development trends in CORBA

Audience

- Software Developers and Managers who are getting started with CORBA development

Duration

- 1 day

Prerequisites

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

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

Format

- Lecture

Material

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

Schedule

- The schedule of this course can be found on our [website](#)

4.3 CORBA Programming with C++

Goals

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

Audience

- Software developers who will be developing distributed applications using CORBA

Duration

- 4 days

Prerequisites

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

Contents

- What is CORBA?
- Interface Definition Language (IDL)
- CORBA Object Overview
- IDL to C++ Mapping Details
- 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

Format

- Lecture and programming exercises

Material

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

Schedule

- The schedule of this course can be found on our [website](#)

4.4 Advanced CORBA Programming with TAO (for Real-Time)

Goals

- Configure TAO for use in a Real-Time Environment
- Configure TAO for implementations requiring highly optimized CORBA solutions
- Use the features of the TAO Real-Time Scheduling Service
- Use the features of the TAO Real-Time Event Service
- Select an appropriate TAO configuration for your domain

Audience

- CORBA developers requiring a highly optimized or Real-Time CORBA implementation. Also, Real-Time developers wanting to take advantage of the distributed object architecture of CORBA

Duration

- 4 days

Prerequisites

- Intermediate CORBA and C++ programming experience. Previous Exposure to Real-Time programming, ACE, and TAO would be helpful

Contents

- TAO as general purpose ORB
- TAO configuration

- What is RT CORBA?
- Meeting the challenges of Real-Time
- TAO Design Goals
- TAO Compile Time Optimizations
- TAO Run Time Optimizations
- TAO Threading models
- Advanced Configuration of TAO
- Choosing a TAO Configuration
- Advanced Building and Debugging Using TAO
- Using the CORBA Messaging specification
- Using Portable Interceptors
- Using TAO's Real-Time Event Service
- Using TAO's Notification Service
- Using TAO's Implementation Repository

Format

- Lecture and hands-on programming exercises

Material

- Each student will get a print out of all the sheets, a printed version of the TAO Programmers Guide from Remedy IT and a copy of the latest version of the TAO Developers Guide from OCI

Schedule

- The schedule of this course can be found on our [website](#)

How to obtain ACE/TAO/CIAO

5

As user of ACE/TAO/CIAO you can download the latest and older versions from <http://download.dre.vanderbilt.edu>. From the main page of this website you can download the latest minor, bug fix only, and beta version. Through this [link](#) you can download any older version. The full version ships with generated GNU makefiles, Visual Studio 7.1/8.0 project files, Borland C++ makefiles, and autoconf support. The source only packages lack the generated makefiles and project files and are therefor much smaller.

5.1 Major release

Major releases are published very infrequently. The last major release (ACE 5.0/TAO 1.0) has been published on July 31, 1999. The next major release will be published when CIAO is mature enough to get a 1.0 version, which is probably not before the end of 2007.

5.2 Minor release

Minor releases are published about once each 12 to 18 months. The last minor release is x.5 which was published March 5, 2006. At this moment the latest beta release has much better support and quality compared to the x.5 version. The x.6 version of ACE/TAO/CIAO is scheduled for summer 2007.

5.3 Bug fix only release

After a major or minor release there is always a bug fix only (BFO) release published. This BFO release only adds bug fixes for ACE and TAO (for CIAO we allow research work to be added). So the x.5.1 version would have some bugs fixed that where found after x.5 was released.

5.4 Beta release

About each 6 to 12 weeks a new beta release is published. In the past these were real beta's, no guarantees for compilation and testing. From around x.4.8 the quality control has been improved a lot. Using the build and test scoreboard at <http://scoreboard.theaceorb.nl> there is now 100% insight in build and test results. A beta is now only released when it compiles on all platforms on the scoreboard and the test results are comparable with any beta in the past. It can be that new tests are failing because they uncover an old bug, but we don't publish a beta that has broken core functionality.

This chapter explains the steps to create your first Hello world application using TAO.

6.1 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}`.

6.2 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 should 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
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.

```
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);

Test::Hello_var hello = hello_impl->_this ();

```

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& ex)
    {
        ex._tao_print_exception ("Exception caught:");
        return 1;
    }

    return 0;
}
```

The server is now ready.

6.3 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 "TestC.h"
```

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
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.

```

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 %s\n",
                      ior),
                      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 %s\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& ex)
{
    ex._tao_print_exception ("Exception caught:");
    return 1;
}

return 0;
}
```

6.4 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 {
        Test.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 {
    TestC.cpp
    TestS.cpp
  }
  IDL_Files {
  }
}

project(*Client): taoclient {
  after += *idl
  Source_Files {
    client.cpp
  }
  Source_Files {
    TestC.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++ 8, you can generate the solution and project files with MPC:

```
$ACE_ROOT/bin/mwc.pl -type vc8
```

On Windows, with Visual C++ 7.1, you can generate the solution and project files with MPC:

```
$ACE_ROOT/bin/mwc.pl -type vc71
```

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

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

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
```

6.5 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.

7.1 Generated Files

The 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 for example, 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 [Table 7.1](#).

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"

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 TAO_IDL 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

Table 7.1 TAO_IDL Environment Variables

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 overridden 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:

- 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
- 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
- Set the environment variable `$ACE_ROOT` appropriately or add `$ACE_ROOT/bin` to your search path
- 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 [Table 7.2](#).

-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

Table 7.2 TAO_IDL Operation Demuxing Strategies

7.4 Collocation Strategies

TAO_IDL can generate collocated stubs using two different collocation strategies. It also allows you to suppress/enable the generation of the stubs of a particular strategy. To gain great flexibility at runtime, you can generate 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 [bugzilla 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 [Table 7.3](#)

-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
-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

Table 7.3 TAO_IDL Output File Options

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 [Table 7.4](#), or to suppress parts that we generate by default but are not required for some applications as listed in [Table 7.5](#).

-GT	Enable generation of the TIE classes, and the *S_T.* files that contain them
-----	--

Table 7.4.a TAO_IDL Additional flags

-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.
-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
-Guc	Generate uninline 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

Table 7.4.b TAO_IDL Additional flags

-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)

Table 7.5.a TAO_IDL Suppression flags

-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 ¹
-Sci	Suppress generation of the client inline file ¹
-Scc	Suppress generation of the client stub file ¹
-Ssi	Suppress generation of the server inline file ¹
-Ssc	Suppress generation of the server skeleton file ¹
-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

Table 7.5.b TAO_IDL Suppression flags

7.7 Backend options

Table 7.4 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-seperated list of the listed backend options.

<code>skel_export_macro=macro_name</code>	The compiler will emit <code>macro_name</code> 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.
<code>skel_export_include=include_path</code>	The compiler will generate code to include <code>include_path</code> at the top of the generated server header, this is usually a good place to define the server side export macro.
<code>stub_export_macro=macro_name</code>	The compiler will emit <code>macro_name</code> 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.
<code>stub_export_include=include_path</code>	The compiler will generate code to include <code>include_path</code> at the top of the client header, this is usually a good place to define the export macro.
<code>anyop_export_macro=macro_name</code>	The compiler will emit <code>macro_name</code> 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 conjunction with the <code>-GA</code> option, which generates Any operators and typecodes into a separate set of files.
<code>anyop_export_include=include_path</code>	The compiler will generate code to include <code>include_path</code> at the top of the anyop file header, this is usually a good place to define the export macro. This option works in conjunction with the <code>-GA</code> option, which generates Any operators and typecodes into a separate set of files.

Table 7.6.a TAO_IDL Backend Options

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 Borland C++Builder or MSVC++.
obv_opt_accessor	The IDL compiler will generate code to optimize access to base class data for valuetypes.
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 Borland 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 Borland 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.

Table 7.6.b TAO_IDL Backend Options

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 [Table 7.7](#).

-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
-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 TMPDIR 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 indentifiers in the same scope differ in spelling only by case (default)

Table 7.7 TAO_IDL Other flags

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 `PortableServer.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.

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 <code>-Sa</code> and <code>-St</code> are removed from the default idl flags.	anytypecode
TAO_BiDirGIOP	Support for BiDirectional GIOP as outlined by the CORBA spec. Please see <code>\$TAO_ROOT/tests/BiDirectional</code> for a simple test case of this feature. Applications need to <code>#include "tao/BiDir_GIOP/BiDirGIOP.h"</code> within their code to get this feature.	bidir_giop

Table 8.1.a List of CORE Libraries in TAO

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 <code>#include "tao/CodecFactory/CodecFactory.h"</code> within their code to get this feature.	codecfactory
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 <code>#include "tao/DynamicAny/DynamicAny.h"</code> 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 <code>#include "tao/EndpointPolicy/EndpointPolicy.h"</code> to get the right symbols.	endpointpolicy
TAO_DynamicInterface	Support for DII and DSI invocations. Applications have to <code>#include "tao/DynamicInterface/Dynamic_Adapter_Impl.h"</code> to get the right symbols.	dynamicinterface
TAO_IFR_Client	Support for client/stub side interfaces for InterfaceRepository applications. Applications have to <code>#include "tao/IFR_Client/IFR_Client_Adapter_Impl.h"</code> 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 <code>#include "tao/ImR_Client/ImR_Client.h"</code> to get the right symbols.	imr_client
TAO_IORInterceptor	Support for IORInterceptor. The portable server library depends on the IORInterceptor library. Applications have to <code>#include "tao/IORInterceptor/IORInterceptor_Adapter_Factory_Impl.h"</code> to get the right symbols.	iorinterceptor

Table 8.1.b List of CORE Libraries in TAO

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 <code>#include "tao/IORManipulation/IORManip_Loader.h"</code> 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 <code>#include "tao/IORTable/IORTable.h"</code> 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 <code>#include "tao/Messaging/Messaging.h"</code> 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 <code>resolve_initial_references ("RootPOA")</code> ; 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 <code>#include "tao/RTCORBA/RTCORBA.h"</code> to get the required symbols for linking. Support in this library is complaint with RTCORBA 1.0 spec.	rt_client

Table 8.1.c List of CORE Libraries in TAO

TAO_RTPortableServer	Support for RTCORBA server side features. Applications are required to <code>#include "tao/RTPortableServer/RTPortableServer.h"</code> to get the required symbols for linking. Support in this library is compliant with RTCORBA 1.0 spec.	rt_server
TAO_RTScheduling	Support for RTCORBA 1.2 features. Applications are required to <code>#include "tao/RTScheduling/RTScheduling.h"</code> to get the required symbols for linking. Support in this library is compliant 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 <code>#include "tao/Strategies/advanced_resources.h"</code> .	strategies
TAO_TypeCodeFactory	Support for TypeCodeFactory interface.	typecodefactory
TAO_Utills	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 <code>#include "tao/CSD_ThreadPool/CSD_ThreadPool.h"</code> for static loading and provide service configuration file for dynamic loading.	csd_threadpool

Table 8.1.d List of CORE Libraries in TAO

TAO_TC	Support for TAO::Transport::Current - a generic framework for applications that need access to statistical information about the currently used Transport. This library can be loaded statically or dynamically. Applications are required to <code>#include "tao/TransportCurrent/Transport_Current.h"</code> 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 <code>#include "tao/TransportCurrent/IIOP_Transport_Current.h"</code> 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 <code>#include "tao/Compression/Compression.h"</code> for static loading.	compression
TAO_ZlibCompressor	Support for Zlib Compression. This library can be loaded statically or dynamically. Applications are required to <code>#include "tao/Compression/zlib/ZlibCompressor.h"</code> for static loading.	zlibcompressor

Table 8.1.e List of CORE Libraries in TAO

Compression 9

Starting with TAO 1.5.5 the compression library exists. With this library it is possible to compress and uncompress 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_compressor` method is the id of the compressor you want. These predefined id's are listed in [Table 9.1](#).

Compression::COMPRESSORID_GZIP	gzip
Compression::COMPRESSORID_PKZIP	pkzip
Compression::COMPRESSORID_BZIP2	bzip2
Compression::COMPRESSORID_ZLIB	zlib

Table 9.1 Compressor Ids

```

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

```

A compressor is capable of compression CORBA::OctetSeq 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.

```
CORBA::OctetSeq 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 `OctetSeq` that can be used to put the decompressed data in, this `OctetSeq` must have a length large enough to contain the decompressed data. At the moment then second `OctetSeq` is not large enough a `Compression::CompressionException` will be thrown. The compressed `OctetSeq` 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.

```
CORBA::OctetSeq 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 algorithm.

```

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 ::CORBA::OctetSeq &source,
    ::CORBA::OctetSeq &target);

virtual void decompress (
    const ::CORBA::OctetSeq &source,
    ::CORBA::OctetSeq &target);
};

```

The constructor just passes the values to its base, this compressor is very easy, it doesn't need to store any additional 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 ::CORBA::OctetSeq &source,
    ::CORBA::OctetSeq &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 ::CORBA::OctetSeq &source,

```

```
    ::CORBA::OctetSeq &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.

Using the TAO::Transport::Current Feature

10

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 `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 (milliseconds) 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 tobject =
    orb->resolve_initial_references ("TAO::Transport::Current");

::TAO::Transport::Current_var tc =
    ::TAO::Transport::Current::_narrow (tobject.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 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.

10.5.1 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 currently 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.

10.5.2 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.

10.6 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& tv);
    const ACE_Time_Value& 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.

10.7 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.

10.8 Example Code

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

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..."` 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 [table 11.1](#).

-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 <code>SecurityLevel2::SecurityManager</code> interface as defined in the CORBA Security Service is implemented.
-SSLCertificate <i>FORMAT:filename</i>	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 <i>FORMAT:filename</i>	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 <i>which</i>	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 <i>which</i>	Set the maximum amount of time to allow for establishing a SSL/TLS passive connection, <i>i.e.</i> for accepting a SSL/TLS connection. The default value is 10 seconds. See the discussion in bugzilla 1348 for the rationale behind this option.
-SSLDHParams <i>filename</i>	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.

Table 11.1 SSLIOP Options

11.3 Environment variables

The SSLIOP protocol supports the environment variables listed in [table 11.2](#) to control its behavior.

SSL_CERT_FILE <i>filename</i>	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.
SSL_CERT_DIR <i>directory</i>	<p>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:</p> <pre>\$ ln -s cacert.pem 'openssl x509 -noout -hash \$lt; cacert.pem'.0</pre> <p>Consult the documentation of your SSL implementation for more details.</p>
SSL_EGD_FILE <i>filename</i>	The name of the UNIX domain socket that the Entropy Gathering Daemon (EGD) is listening on.
SSL RAND_FILE <i>filename</i>	The file that contains previously saved state from OpenSSL's pseudo-random number generator.

Table 11.2 SSLIOP Environment Variables

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, &der_cert, cert->length ());
::X509_NAME_oneline (
    ::X509_get_issuer_name (peer),
    buf,
    BUFSIZ);

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

::X509_free (peer);
```

Real Time CORBA 12

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 the TPG.

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`

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

Table 12.1 IIOP Protocol Properties

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

long send_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
long recv_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ

Table 12.2 UIOP Protocol Properties

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

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

Table 12.3 SHMIOP Protocol Properties

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

long send_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
long recv_buffer_size	ACE_DEFAULT_MAX_SOCKET_BUFSIZ
enable_network_priority	false

Table 12.4 DIOP Protocol Properties

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

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

Table 12.5 SCIOP Protocol Properties

12.1.6 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 Protocol-Properties 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.

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 [OMG specification ptc/2006-08-03](#).

13.1.1 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.

13.1.2 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.

13.2 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 [Chapter 5](#)). Then you can add `corba_e_compact=1` or `corba_e_micro=1` to the `default.features` file and regenerate the makefiles using MPC.

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++ Network Programming: Mastering Complexity Using ACE and Patterns

C++NPv1 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++ Network Programming: Systematic Reuse with ACE and Frameworks

C++NPv2 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 underly 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 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.

CORBA Books 15

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.

Frequently asked questions

16

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 in progress and scheduled for completion Q2 of 2007.

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.

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. Because FC6/FC7/RH5 have integrated the fix but SuSE has not integrated the same fix in their GCC 4.1.1 we can't automatically disable the workaround. You have to do that 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 ACED_DEUBG macro use the following in your config.h file and recompile ACE.

```
#define ACE_NDEBUG 0
```

To disable it:

```
#define ACE_NDEBUG 1
```

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.

I want to use Microsoft Visual C++ 9.0 beta but do get compile errors in intrin.h on _interlockedbittestandset and _interlockedbittestandreset.

This is a known problem already reported to Microsoft. Open the intrin.h that gets shipped with MSVC9 and comment out these two methods.

What is the latest version of ACE/TAO/CIAO that is maintained for VxWorks 5.5.x?

The latest version is x.5.9. After this beta we did end the daily maintenance due to the lack of funding for this effort.

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

17.1 Building with Microsoft Visual C++

ACE contains project files for Microsoft Visual Studio .NET 2003 (VC7.1) and Visual Studio 2005 (VC8). Visual Studio 2005 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, VC7.1 and VC8 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 [Table 17.1](#). All solution files have a .sln suffix and all project files have a .vcproj suffix.

VC7.1	name (no decorator)
VC7.1 static	name_Static
VC8 for desktop/server	name_vc8
VC8 for Windows CE/Mobile	name_WinCE

Table 17.1 MSVC Solutions

If you happen to open a VC7.1 file from within VC8, it will offer to convert the file to the newer format for you. With the stock VC8, do not do this; Visual Studio will crash while attempting to convert the large solution and project files to build ACE. Simply refuse the conversion and open the file with the correct format. Note that Microsoft has fixed this problem. See [MSDN](#) for information.

1. Uncompress 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 uncompressed into the root directory.
2. Create a file called `config.h` in the `ACE_ROOT\ace` directory that contains:

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

3. The static, DLL and MFC library builds are kept in different workspaces. Files with names `*_Static` contain project files for static builds. Workspaces for static and DLL builds will be available through the stock release at DOC group's website. The workspaces for MFC are not available and have to be generated using MPC. Please see MPC's README for details.
4. Now load the solution file for ACE (`ACE_ROOT/ACE.sln`).
5. 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 NT. 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`.
6. If you want to use the standard C++ headers (`iostream`, `cstdio`, ... as defined by the C++ Standard Draft 2) 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`.

7. 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 NT box 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.

8. 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.)

9. 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"}

17.2 Building with GNU make

When you use GNU make you can use two methods of building ACE:

1. Traditional ACE/GNU Make Configuration
2. GNU Autoconf

17.2.1 Traditional per-platform configuration method

Here's what you need to do to build ACE using GNU Make and ACE's traditional per-platform configuration method:

1. 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.
2. 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, if also CIAO is build then also CIAO_ROOT has to be set.

3. 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 [Table 17.2](#) 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.

4. 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 [Table 17.3](#) for an overview of the available platform files.

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

5. 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
```

6. 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`.

config- <code>aix-5.x.h</code>	AIX 5
--------------------------------	-------

Table 17.2.a Available config files

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

Table 17.2.b Available config files

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

Table 17.3.a Available platform files

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
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

Table 17.3.b Available platform files

17.2.2 Building ACE with GNU Autoconf

From x.5.7 GNU Autoconf support is available in all the distributions at the DOC group website, earlier versions only ship autoconf support in the ACE and ACE+TAO distribution. GNU Autoconf support has been partially present in a number of ACE versions. However, ACE 5.4 was the first version that supported it in earnest. The range of platforms on which GNU autoconf support is regularly tested is very limited compared with the traditional configuration method, so you should be careful to test the resulting ACE library before using it in your applications. You can review the build scoreboard to check the currently tested set of autoconfigured platforms (look for autoconf in the platform name). Any help you can lend to improve the ACE build process using GNU Autoconf would be very much appreciated. Please send any fixes to the ACE users mailing list using the standard PROBLEM-REPORT-FORM.

The kit has been bootstrapped so you do not need to install the GNU Autotools (autoconf, automake, libtool) unless you want to participate in testing and developing this process further or if you are working directly off of sources in the ACE subversion repository. To simply configure and build ACE, do:

- cd to the top-level ACE_wrappers directory.
- Create a subdirectory to hold your build's configuration and built ACE version, and then change to the new directory:


```
mkdir build
cd build
```

Note that you do not run the `create_ace_build.pl` utility mentioned in the Cloning the Source Tree section. The configure script takes care of creating all files and links that are needed.

- Configure ACE for your platform by issuing the following command:

```
../configure [options]
```

options can be a variable setting (such as setting `CXX` to your C++ compiler command) any standard GNU configure options, or any of the following ACE configure options (default values are in parentheses, prefix with `--`):

- `enable-alloca` (no): Enable `alloca()` support.
- `enable-debug` (yes): Build ACE with debugging support.
- `enable-exceptions` (yes): Build ACE with C++ exception support compiled in.
- `enable-fast` (no): Use the Sun C++ `-fast` option to build. Only used on Solaris.
- `enable-ipv4-ipv6` (no): Enable IPv4/IPv6 migration support.
- `enable-ipv6` (no): Enable IPv6 support.
- `enable-inline` (yes): Enable inline functions.
- `enable-optimize` (yes): Enable building optimized.
- `enable-prof` (no): Enable profiling support.
- `enable-purify` (no): Build with support for IBM Rational Purify.
- `enable-quantify` (no): Build with support for IBM Rational Quantify.
- `enable-repo` (no): Enable the GNU g++ `-frepo` option. Only useful for pre-3.0 g++.
- `enable-rtti` (yes): Compile with C++ runtime type information RTTI support.
- `enable-stdcplib` (yes): Build with support for the standard C++ library, as opposed to the older `iostreams` library.
- `enable-log-msg-prop` (yes): Enable `ACE_Log_Msg` property propagation to ACE-created threads.
- `enable-logging` (yes): Enable the ACE logging macros.
- `enable-malloc-stats` (no): Compile in additional code for collecting memory allocation statistics.
- `enable-pi-pointers` (yes): Enable position-independent pointers for shared memory classes.
- `enable-probe` (no): Enable the `ACE_Timeprobe` class.

- enable-reentrant (yes): Enable use of platform’s reentrant functions.
- enable-static-obj-mgr (yes): Enable use of a static ACE_Object_Manager.
- enable-threads (yes): Enable threading support.
- enable-verb-not-sup (no): Enable verbose ENOTSUP reports at run time.
- enable-trace (no): Enable ACE execution tracing support.
- enable-fl-reactor (no): Enable support for the ACE_FlReactor class.
- enable-qt-reactor (no): Enable support for the ACE_QtReactor class.
- enable-tk-reactor (no): Enable support for the ACE_TkReactor class.
- enable-xt-reactor (no): Enable support for the ACE_XtReactor class.
- enable-gperf (yes): Build the implementation of gperf that comes with ACE.
- enable-qos (no): Include the ACE_QoS library when building ACE.
- enable-ssl (yes): Include the ACE_SSL library when building ACE. Requires the SSL components to be available using the compiler’s and linker’s default search directories.
- with-openssl: Specifies the root directory of the OpenSSL installation; expects the specified directory to have include and lib subdirectories. To specify other locations for the header and libraries, use one or both of the following.
- with-openssl-include: Specify the directory containing the OpenSSL header files.
- with-openssl-libdir: Specify the directory containing the OpenSSL libraries.
- with-tli-device (/dev/tcp): Specifies the device name for opening a TLI device at run time.
- Build ACE by typing make.
- (Optional) Install ACE by typing make install.

Testing and Developing GNU Autotool Support in ACE

In order to test and develop the GNU Autotool support in ACE or bootstrap autotool support into ACE when working directly off of ACE sources in the subversion repository, you must have recent versions of GNU Autoconf, Automake and Libtool installed on your host. Once installed, autotool support may be bootstrapped into your workspace by doing the following:

```
cd ACE_wrappers
./bin/bootstrap
```

After doing so, you will be able to run the configure script.

17.3 Building with C++ Builder

Before building ACE/TAO/CIAO you should check your C++ Builder installation to see if it is supported. At this moment the following C++ Builder versions are supported:

- Borland C++ Builder 6 with Update Pack 4 installed
- Borland C++ Builder 2006 with Update Pack 2 installed
- CodeGear C++ Builder 2007

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

17.3.1 Building the libraries

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

1. Uncompress 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 if you uncompressed into the root directory.
2. Create a file called config.h in the ACE_ROOT\ace directory that contains:

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

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

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

5. Set the BCBVER environment variable to the main version of your C++ Builder compiler. 6 is the value for Borland C++ Builder 6, 8 the value for Borland C++ Builder 2006 and CodeGear C++ Builder 2007.

```
set BCBVER=6
```

6. Change to the ACE_ROOT\ace directory.
7. Build release DLLs for ACE by doing:

```
make -f Makefile.bor all
```

The make should be the Borland make, not the GNU make utility. You can set additional environment variables to build a different version of ACE, see [Table 17.4](#) for an overview of the supported environment variables.

8. 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.

DEBUG=1	Build a debug version
STATIC=1	Build a static version
UNICODE=1	Build an unicode version
UNICODE=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 Borland help for more info.

Table 17.4 Environment Variables

17.3.2 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.bor 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.bor 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.

17.3.3 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 codeblock 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)
{
    // ...
```

```
}
```

17.4 Building a host build

Cross compilation is a setup where you build TAO/CIAO 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 when you only use TAO and additional tao_idl3_to_idl2 and cidlc when you also use CIAO. 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 [Table 17.2](#) 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 [Table 17.3](#) 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 only 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
  apps/gperf/src
  TAO/TAO_IDL
}
```

When we are also using CIAO we create the file `$ACE_ROOT/CIAO_Host.mwc` with the contents:

```
workspace {
  ace
  apps/gperf/src
  TAO/TAO_IDL
  TAO/CIAO/IDL3_to_IDL2
}
```

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
perl bin/mwc.pl CIAO_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.

If you are using CIAO, `cidlc` is a different story. This compiler needs boost to be build and takes more to setup. The easiest is to download a prebuild version from the webserver at [Vanderbilt University](#).

17.5 Building for 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 [Section 17.4](#) 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 do 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
TAO_IDL := $(HOST_ROOT)/bin/tao_idl
TAO_IDL_DEP := $(TAO_IDL)
TAO_IDLFLAGS += -g $(HOST_ROOT)/bin/gperf
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.

17.5.1 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.

17.5.2 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"), NAME))
#define ACE_END_TEST ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("%P|%t) Ending %s test t %D"), program));
```

17.5.3 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.

17.5.4 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 <http://bochs.sourceforge.net/getcurrent.html>.

17.6 Building from the subversion repository

July 2006 the ACE/TAO/CIAO cvs repository at Washington University was replaced with a subversion repository (snv) at Vanderbilt University. With this move it is now possible as user of ACE/TAO/CIAO to checkout a version from the repository and have a look at work in progress. An important note is that the doc_group doesn't advice to use the svn 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.

The first step is to determine whether you want to build ACE, ACE+TAO or ACE+TAO+CIAO. For building you will always need MPC. Based on the subset there is a different set you have to checkout from the subversion repository. For checking out ACE the subversion command to use:

```
svn co svn://svn.dre.vanderbilt.edu/DOC/Middleware/sets-anon/ACE .
```

For ACE+TAO:

```
svn co svn://svn.dre.vanderbilt.edu/DOC/Middleware/sets-anon/ACE+TAO .
```

For ACE+TAO+CIAO

```
svn co svn://svn.dre.vanderbilt.edu/DOC/Middleware/sets-anon/ACE+TAO +CIAO .
```

For MPC

```
svn co svn://svn.dre.vanderbilt.edu/DOC/MPC .
```

For TAO DDS

```
svn co svn://svn.dre.vanderbilt.edu/DOC/DDS .
```

In the subversion repository no GNUmakefiles and project files for building on the various platforms are available. If you build from subversion you have to generate these makefiles before building ACE/TAO/CIAO. 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.mwc
```

On Windows, with Visual C++ 8, you must generate solution and project files with MPC:

```
$ACE_ROOT/bin/mwc.pl -type vc8 ACE.mwc
```

On Windows, with Visual C++ 7, you must generate solution and project files with MPC:

```
$ACE_ROOT/bin/mwc.pl -type vc71 ACE.mwc
```

On Windows, with Borland C++, you must generate solution and project files with MPC:

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

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

For ACE/TAO/CIAO we have created an autobuild framework that can be used to build ACE/TAO/CIAO 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.

As extension there is a integrated test scoreboard that stores all autobuild data into a database on makes it available through a website (see <http://scoreboard.theaceorb.nl> for an example). This extension is not public available yet.

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

- Perl ([ActiveState Perl](#) for Windows hosts)
- [Subversion](#) client (svn)
- [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
svn co svn://svn.dre.vanderbilt.edu/ACE/autobuild/trunk 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 different settings you have (for starting point see autobuild/configs/autobuild/tango_isis/Debian_Core.xml)

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`, `TAO_ROOT` and `CIAO_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="CIAO_ROOT" value="/build/ACE/ACE_wrappers/TAO/CIAO"/>
<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,TAO/CIAO/CIAO_*.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>
```

19.1 Changes in 5.5.9/1.5.9/0.5.9

19.1.1 ACE 5.5.9

- Use Intel C++ specific optimizations for Linux on IA64
- Improved support for `ACE_OS::fgetc`. Added support for `ACE_OS::fputc`, `ACE_OS::getc`, `ACE_OS::putc` and `ACE_OS::ungetc`.
- Added support for `ACE_OS::log2(double)` and improved support for `ACE::log2(u_long)`.
- Shared library builds on AIX now produce a `libxxx.so` file instead of the previous practice of producing `libxxx.a(shr.o)`.
- GCC 4.1.2 that comes with Fedora 7 seems to have a fix for the visibility attribute we use for the singletons. F7 users will therefore need to define the following in your `config.h` file: `ACE_GCC_HAS_TEMPLATE_INSTANTIATION_VISIBILITY_ATTRS 1`
- Fixed (rare) problem in `TP_Reactor` where incorrect event handler was resumed.
- Reduced footprint on some platforms, particularly those that use `g++ >= 3.3`.

19.1.2 TAO 1.5.9

- When using AMI collocated in case of exceptions they are delivered to the reply handler instead of passed back to the caller.
- Added support for IPv6 multicast addresses when federating `RTEvent` channels.
- Fixed a bug in the IDL compiler's handling of octet constants where the rhs consists of integer literals and infix operators ([bugzilla 2944](#)).
- `TAO_IDL` enhancements for built in sequence support in TAO DDS.
- Provide support for TAO DDS zero-copy read native types.
- `TAO_IDL` fix for lock-up and incorrect error message generation for missing definitions within local modules. This fix also stops the erroneous lock-up in a parent scoped module (with the same name as a locally scoped module) that should have been hidden by the local scope definition.

- The TAO_IORManip library now has a filter class that allows users to create new object references based on existing multi-profile object references by filtering out profiles using user defined criteria. The use of `-ORBUseSharedProfile 0` is required for this to function.
- A problem in `TAO_Profile::create_tagged_profile` was fixed. This problem triggered only when MIOP in multi-threaded application was used.
- Added IPv6 support in MIOP so that IPv6 multicast addresses can be used in addition to those IPv4 class D addresses. DSCP support is implemented in MIOP as well. However, since MIOP only allows one way communication then it makes sense to use DSCP on the client side only.

19.1.3 CIAO 0.5.9

- Added a new deployment algorithm to DAnCE for optimization of large scale deployment. In this algorithm, the number of threads spawned is based on the deployment plan, i.e, by parsing the deployment plan information, DAnCE decides how many threads to spawn (one thread per node). This algorithm knows how to "initialize" each thread since each thread will have different execution context, i.e., which components to deploy to which node, component configuration, etc.
- Added a NA component server callback wait strategy in NAM, which uses conditional variable to signal the NA call back so it can work with multi-threaded configuration, such as thread-per-connection mode. The original implementation uses the main thread to run the ORB event loop, which will not work for multi-threaded environment.

19.2 Changes in 5.5.8/1.5.8/0.5.8

19.2.1 ACE 5.5.8

- Extended `ACE_Event` constructor with optional `LPSECURITY_ATTRIBUTES` argument
- Added support for QT4
- Added support to integrate with the [FOX Toolkit](#)
- Added support for Microsoft Visual Studio Code Name "Orcas", which is the msvc9 beta
- Added ability to provide an optional priority when calling `ACE_Message_Queue_Ex::enqueue_prio()`. There was previously no way to specify a priority for queueing

- Removed support for Visual Age on Windows
- ACE will compile once again with `ACE_LACKS_CDR_ALIGNMENT` #defined
- `ACE_Process_Manager::terminate()` no longer removes the process from the process descriptor table; the pid remains available in order to call `ACE_Process_Manager::wait()`

19.2.2 TAO 1.5.8

- Fixed bug in IDL compiler related to abstract interfaces
- Fixed several issues in the AMI support
- Added new `-ORBAMICollocation 0` which disables AMI collocated calls
- Improved a lot of test scripts to work with a cross host test environment where client and server are run on different hosts. This is used for automated testing with VxWorks
- Fixed handled of a forward request when doing a locate request call
- Added an option, `-a`, to the `Event_Service` to use the thread-per-consumer dispatching strategy instead of the default dispatching strategy.
- Improved wide character compilation support.
- Fixed IDL compiler to run on both OpenVMS Alpha and OpenVMS IA64.
- Fixed memory leaks in the `ImpleRepo_Service`.
- Fixed a bug in the IDL compiler relating to include paths.
- Fixed Trader Service issues related to `CORBA::Long` and `CORBA::ULong`.

19.2.3 CIAO 0.5.8

- Improved the option handling of the `Execution_Manager` and `plan_launcher`.
- Added a utility library to manipulate the deployment plan, such as adding/removing instances, adding/removing connections, and pretty print.

19.3 Changes in 5.5.7/1.5.7/0.5.7

19.3.1 ACE 5.5.7

- ACE 5.5 contained a set of pragmas which prevented Visual Studio 2005 (VC8) from issuing warnings where C run-time functions are used but a more secure alternative is available. For more information on the C run-time issues and Microsoft's response, please see [MSDN](#). In this beta, the pragmas which prevented the warnings have been removed. The ACE library has been reviewed and most of the use of "unsafe" functions has been fixed where possible. Since not all of the warnings emanating from ACE are situations that can or should be fixed, the ACE VC8 projects will prevent the warnings while building the ACE kit and its contained examples, tests, etc. The warnings are disabled by adding Microsoft-specified macros to the compile line via MPC. If desired, the warnings can be re-enabled by regenerating the project files with different MPC features. Note, however, that while ACE without warnings caused by the new C run-time functions, your application builds may trigger these warnings either by use of the "unsafe" C run-time functions or via use of an inlined ACE_OS method which uses it. If the warning is caused by an ACE_OS method, there is a more safe alternate available, probably located by appending _r to the method name (e.g., instead of using ACE_OS::ctime(), use ACE_OS::ctime_r()). There are other cases where the compiler may have issued warnings and ACE prevented this via a #pragma. These #pragmas have been removed as well. This may cause your application builds to trigger more warnings from VC8 than past ACE versions. You should review your code and either correct the code or disable the warnings locally, as appropriate.
- The "release" argument to a number of ACE_String_Base<> methods was changed from int to bool to more accurately reflect its purpose. The following methods were changed:

```
ACE_String_Base (const CHAR *s, ACE_Allocator *the_allocator = 0, int release = 1);
```

to

```
ACE_String_Base (const CHAR *s, ACE_Allocator *the_allocator = 0, bool release = true);
```

```
ACE_String_Base (const CHAR *s, size_type len, ACE_Allocator *the_allocator = 0, int release = 1);
```

to

```
ACE_String_Base (const CHAR *s, size_type len, ACE_Allocator *the_allocator = 0, bool release = true);
```

```
void set (const CHAR * s, int release = 1);
```

to

```
void set (const CHAR * s, bool release = true);
void set (const CHAR * s, size_type len, int release);
```

to

```
void set (const CHAR * s, size_type len, bool release);
void clear (int release = 0);
```

to

```
void clear (bool release = false);
```

Since `ACE_String_Base` forms the basis of the `ACE_CString` and `ACE_TString` classes, this may ripple out to user application code. If you encounter errors in this area while building your applications, replace the `int` argument you are passing to the method now with either `true` or `false`.

- Solutions for the eVC3/4 platform have been removed from this release. Note that we package WinCE projects/workspaces for use with VC8.
- There were 3 new `ACE_Log_Msg` logging format specifiers added to make logging easier for types that may change sizes across platforms. These all take one argument, and the new formats are:

```
%b - format a ssize_t value
%B - format a size_t value
%: - format a time_t value
```

- The `ace/Time_Request_Reply.h` and `ace/Time_Request_Reply.cpp` files were moved from `$ACE_ROOT/ace` to `$ACE_ROOT/netsvcs/lib`. The time arguments in the public API to `ACE_Time_Request` were changed from `ACE_UINT32` to `time_t` and the portions of the on-wire protocol that contains time was changed from `ACE_UINT32` to `ACE_UINT64`. Thus, code that uses the `ACE_Time_Request` class to transfer time information will not interoperate properly with prior ACE versions. This will affect uses of the `netsvcs` time clerk/server.
- The portion of the `ACE_Name_Request` class that carries the on-wire seconds portion of a timeout value was changed from `ACE_UINT32` to `ACE_UINT64`. This means that Name server/clients at ACE 5.5.7 and higher will not interoperate properly with previous ACE versions' name servers/clients.

- In the `ACE_Log_Record` (`ACE_Log_Priority`, `long`, `long`) constructor, the second argument, `long time_stamp`, was changed to be of type `time_t`. This aligns the type with the expected value, a time stamp such as that returned from `ACE_OS::time()`.
- Added support for VxWorks 6.x cross compilation using a Windows host system
- Added support for VxWorks 6.x using the diab compiler
- The destructor of `ACE_Event_Handler` no longer calls `purge_pending_notifications()`. Please see [bugzilla 2845](#) for the full rationale.

19.3.2 TAO 1.5.7

- Removed `ACE_THROW_RETURN`
- Fixed a memory crash problem when using ETCL IN operator with Notify Service filter.
- Remove exception specifications from ORB mediated operations (C++ mapping requirement)
- New `diffserv` library to specify `diffserv` priorities independent of `RTCORBA`
- Addressed Coverity errors in core TAO libraries, TAO_IDL compiler, stubs and skeletons generated by TAO_IDL and the TAO Notification Service.
- Extended current `DynamicInterface` to allow `DII+AMI+DSI+AMH`.

19.3.3 CIAO 0.5.7

- Removed `ACE_THROW_RETURN`
- Remove exception specifications from ORB mediated operations (C++ mapping requirement)
- All DAnCE core idl files are moved to `DAnCE/Deployment`
- `QoS4CCM` IDL files are moved to `ciao/extension`
- `RACE` has been fully removed from the distribution, a new version is work in progress and will be added again to the distribution when it is ready
- MPC base projects that contained `_dnc_` have been renamed to not include that string.
- DAnCE executables are now installed into `$CIAO_ROOT/bin`

19.4 Changes in 5.5.6/1.5.6/0.5.6

19.4.1 ACE 5.5.6

- The `ACE_TYPENAME` macro has been added to those that are not available when the `ACE_LACKS_DEPRECATED_MACROS` config option is set (it is not set by default). You are encouraged to replace the use of `ACE_TYPENAME` with the C++ `typename` keyword before the `ACE_TYPENAME` macros is removed from ACE in the future.
- A new script, `rm_exception_macros.pl`, has been added to help users remove the use of the ACE exception macros from their own code.

19.4.2 TAO 1.5.6

- Removed all exception environment macros except `ACE_THROW_RETURN` and `ACE_THROW_SPEC`

19.4.3 CIAO 0.5.6

- Removed all exception environment macros except `ACE_THROW_RETURN` and `ACE_THROW_SPEC`
- All CIAO libraries built on UNIX systems will now have the correct library version numbers. Previously they had the same version numbers as TAO libraries.

19.5 Changes in 5.5.5/1.5.5/0.5.5

19.5.1 ACE 5.5.5

- The prebuild MPC keyword is now supported by the `gnuace` project type. This fixes [bugzilla 2713](#).
- Support for Windows earlier than NT 4 SP2 was removed. ACE will not build for Windows 95, 98, Me, etc. out of the box any longer.
- Reformat stringified IPv6 addresses to use `[addr]:port` when printing addresses that contain ':' such as `:::1`.
- Added method to `ACE_INET_Addr` to determine if address is IPv6 or IPv4 multicast.

- Fixed a bug in `ACE_Async_Timer_Adapter_Timer_Queue_Adapter<TQ>` where the `gettimeofday` function of the timer queue was ignored when setting the alarm.
- Fixed a problem where, on Solaris 9 onwards, calling `ACE_OS::thr_create(THR_NEW_LWP)` more than 2¹⁵ (65535) times in a process will fail. See changelog entry from "Wed Jan 3 22:31:05 UTC 2007 Chris Cleeland @lt;cleeland_c@ociweb.com>" for more information.
- Fixed a bug in `ACE_QtReactor` where the two `select()` calls in that function might select on different handler sets.
- `ACE_SOCKET_IO::recvv(iovec[], size_t, const ACE_Time_Value* = 0)` and `ACE_SOCKET_IO::sendv(const iovec[], size_t, const ACE_Time_Value* = 0)` methods were changed to specify the `iovec` count argument as `int` instead of `size_t` since it gets reduced to `int` in the underlying OS calls (usually).
- The following deprecated methods were removed:

```
ssize_t ACE_SOCKET_IO::recv (iovec iov[],
                             size_t n,
                             const ACE_Time_Value *timeout = 0) const;

ssize_t ACE_SOCKET_IO::recv (iovec *io_vec,
                             const ACE_Time_Value *timeout = 0) const;

ssize_t ACE_SOCKET_IO::send (const iovec iov[],
                             size_t n,
                             const ACE_Time_Value *timeout = 0) const;
```

These were previously replaced with more specific `recvv()` and `sendv()` methods.

- The 'ignore_suspended' parameter was changed from `int` to `bool` to reflect it's purpose as a yes/no indicator.

```
The ACE_Service_Repository::find(const ACE_TCHAR name[],
                                const ACE_Service_Type **srp = 0,
                                int ignore_suspended = true) const
```

- Added `--enable-ace-reactor-notification-queue` configure script option to the autoconf build for enabling the Reactor's userspace notification queue (defines `ACE_HAS_REACTOR_NOTIFICATION_QUEUE` in `config.h`).
- The `int ACE_OutputCDR::consolidate(void)` method was added. This method consolidates any continuation blocks used by an `ACE_OutputCDR` object into a single block. It's useful for situations which require access to a single memory area containing the encoded stream, regardless of its length, when the length cannot be known in advance.

- There are a number of new methods defined on `ACE_String_Base<CHAR>`:

```
size_t capacity (void) const
```

This method returns the number of allocated CHAR units in the string object.

```
void fast_resize (size_t)
```

This method manage the sizing/reallocating of the string, but doesn't do the memory setting of `resize()`.

```
bool operator!= (const CHAR *) const
bool operator== (const CHAR *) const
```

These methods compare the string with a nul-terminated CHAR* string.

```
nonmember functions operator== and operator!=
```

where also added that compare `const ACE_String_Base` and `const CHAR*`; these make it possible to switch `ACE_String` and `CHAR*` on either side of the operator.

- There are 2 new build options on the traditional make command: `dmalloc` and `mtrace`. When specified at build time (e.g. `make mtrace=1`) the `PLATFORM_DMALLOC_CPPFLAGS` and/or `PLATFORM_MTRACE_CPPFLAGS` values are added to `CPPFLAGS`. For `dmalloc`, the `PLATFORM_DMALLOC_LDFLAGS` and `PLATFORM_DMALLOC_LIBS` are added to `LDFLAGS` and `LIBS`, respectively.
- Added the ability to specify additional purify and quantify command-line options by setting `PLATFORM_PURIFY_OPTIONS` and `PLATFORM_QUANTIFY_OPTIONS`, respectively.
- Added the ability to use [trio](#) if platform lacks decent support for `vsnprintf`. `trio` support is enabled by defining `trio=1` in `platform_macros.GNU`
- Removed Irix 5, DGUX, and m88k support
- Improved LynxOS 4.2 support
- VxWorks 6.4 support
- Added support for FC6. Because the GCC 4.1.1 version that gets shipped has a fix for the visibility attribute we use for the singletons you will need to define the following in your `config.h` file. This can't be done automatically because SuSE 10.2 gets shipped with GCC 4.1.2 but doesn't have the same fix

ACE_GCC_HAS_TEMPLATE_INSTANTIATION_VISIBILITY_ATTRS 1

- RTEMS port

19.5.2 TAO 1.5.5

- Added an IDL compiler option to generate an explicit instantiation and export of template base classes generated for IDL sequences, sometimes necessary as a workaround for a Visual Studio compiler bug. ([bugzilla 2703](#)).
- Beefed up error checking in the IDL compiler when processing `#pragma` version directives.
- Modified IDL compiler's handling of a syntax error to eliminate the chance of a crash ([bugzilla 2688](#)).
- Fixed a bug in code generation for a valuetype when it inherits an anonymous sequence member from a valuetype in another IDL file.
- Extended tests in tests/IDL_Test to cover generated code for tie classes.
- Modified `tao_idl` to emit code to set the exception data in the `Messaging::ExceptionHolder` in the AMI `_except` operation. This has the effect of allowing user defined exceptions to be recognized when raising an exception without collocation. This fixes [bugzilla 2350](#).
- Added hooks to enable custom Object to IOR conversion or allowing local objects (such as Smart Proxies) to be converted to an IOR string.
- Removed warning issued when using `corbaloc` with a default object key.
- Added implementation of Dynamic Any methods `insert_*_seq()` and `get_*_seq()` (spec-defined for sequences of IDL basic types), as well as implementation of `insert` and `get` methods for abstract interfaces.
- Added support for CORBA/e compact
- Added support for CORBA/e micro
- Fixed issues relating to the CosTrading Server library. The constraint language lexer now allow negative floating point values, 64-bit signed and unsigned integers (which can currently be represented as octal or decimal). Also, fixed a bug where negative integers were being stored and compared as unsigned integers which resulted in `-3 > 0` evaluating to true.
- Added Compression module that delivers the infrastructure classes with which data can be compressed. This can be used by regular applications but then also by the ORB in the future.
- Removed support for `-Ge 0` and `-Ge 1` from the IDL compiler. In practice this means that the IDL compiler doesn't generate any environment macros anymore.
- Fixed a problem where TAO mistakenly considered ALL messages with zero-length payload to be errors and was thus not properly parsing and handling the GIOP `CloseConnection` message. This is tested via `Bug_2702_Regression` ([bugzilla 2702](#)).

- Added an optimization to servant activation to eliminate calls to `check_bounds()` on the object key sequence. This has been observed to yield a 30% decrease in activation time for debug builds on VC71 and linux gcc.
- Merged in changes from OCI's distribution which originate from OCI request tickets [RT 8449] and [RT 8881]. In their totality, these changes add a feature whereby the notification service implementation can utilize a separate ORB for dispatching events to consumers.
- Contributed the `Transport::Current` support - a TAO-specific feature providing IDL interfaces which enables users to obtain information about the Transports used to send or receive a message. The basic intent is to provide (typically) a read-only interface to obtaining data like the number of bytes transferred or the number of messages sent and received. Since specific Transports may have very different characteristics, a simple generic implementation for `Transport::Current` is insufficient. This implementation also provides support for specific Transport implementations. See the `TC_IIOp` implementation, which is an IIOp-specific `Transport::Current`. It extends the generic interface with operations providing information about IIOp endpoints, like host and port. By default, TAO builds with support for this feature. Define "`transport_current=0`" in your `default.features` file to disable it. For more details of how the feature is intended to be used, see `docs/transport_current/index.html`

19.5.3 CIAO 0.5.5

- Fixed problems and added command line options to the `generate_component_mpc.pl` Perl script. Also added an HTML documentation file for this script.
- All IDL has been refactored to get a smaller footprint

19.6 Changes in 5.5.4/1.5.4/0.5.4

19.6.1 ACE 5.5.4

- Added appropriate `intptr_t` and `uintptr_t` typedefs on platforms that don't provide them (i.e. when `ACE_LACKS_INTPTR_T` is defined).
- Added ability to explicitly choose support for 32 bit or 64 bit file offsets on all platforms. Define the `_FILE_OFFSET_BITS` preprocessor symbol to either 32 or 64 to choose the desired number of file offset bits. This preprocessor symbol is supported natively by most UNIX and UNIX-like operating systems, and supported by ACE on Windows. Use the new `ACE_OFF_T` typedef to refer to file offsets across UNIX and Windows portably.
- 64-bit file offsets are now enabled by default in Win64 configurations.

- Improved support for 64 bit platforms (64 bit addresses, etc).
- Added STL-style traits, iterators and a `swap()` method to the `ACE_Array_Base<>` class template.
- Added STL-style traits and iterator accessors to the `ACE_Hash_Map_Manager_Ex<>` class template, as well as new `find()` and `unbind()` methods that return (as an "out" parameter) and accept iterators, respectively.
- Greatly improved event handler dispatch performance in `select()`-based reactors (e.g. `ACE_Select_Reactor` and `ACE_TP_Reactor`) for large handle sets on Windows. Previous event handler search were linear, and are now constant on average.
- Addressed a number of Coverity errors (`CHECKED_RETURN`, `DEADCODE`, `LOCK`, `USE_AFTER_FREE`, `RESOURCE_LEAK`, `FORWARD_NULL`).
- Added STL-style "element_type" trait to all ACE auto_ptr class templates.
- Removed support for LynxOS 3.x.
- Resolved [bugzilla 2701](#) to ensure `fini()` is called for all Service Objects upon calling `ACE_Service_Config::close()`
- VxWorks 5.5.2 has been tested, for ACE the support is exactly the same as for VxWorks 5.5.1. No specific defines or flags have to be used.

19.6.2 TAO 1.5.4

- Added support for `ACE_Dev_Poll_Reactor` to `Advanced_Resource_Factory`.
- Improved `tao_idl` performance, particularly over networked filesystems.
- Added new option for the RTEC, `-ECDispatchingThreadsFlags`, that allows the user to pass in a list of thread creation flags and priority for dispatching threads. These can be used for either the MT dispatching strategy or the TPC dispatching strategy. See [docs/ec_options.html](#) for usage information. Also added `-ECDebug` option to enable debugging output from the RTEC. Only the option and variable was added, but no messages. Therefore, at the moment, this does not generate much output.
- Resolved [bugzilla 2651](#) to eliminate incompatibility with the new mechanism, allowing per-ORB configurations.
- Fixed [bugzilla 2686](#), which involved correctly managing memory during exceptional situations. Throwing an exception during the creation of the Root POA would cause a leak of a `TAO_Adapter` and POA manager related objects.
- Fixed [bugzilla 2699](#), by unInlining generated code for the TIE template classes. Inlining of virtual functions in this code was causing problems with RTTI on some compilers. As a side result, the `idl` compiler doesn't generate a `S_T.inl` file anymore.
- Fixed a bug where calling `_set_policy_overrides()` on a collocated servant would return an unusable object reference.
- Addressed a number of Coverity errors (`CHECKED_RETURN`, `DEADCODE`, `LOCK`, `USE_AFTER_FREE`, `RESOURCE_LEAK`, `FORWARD_NULL`). In particular, missing return value checks and unreachable code in the ACE CDR stream implementation were addressed. Memory and resource management in the ACE Configuration classes was corrected. A potential deadlock upon error was fixed in `ACE_OS::rw_unlock()`. Missing

return value checks were addressed in `ACE_OS::open()` on Windows and `ACE_Thread_Manager::wait()`. A potential dereference of a null pointer in `ACE_OS::scandir_emulation()` was corrected. Lastly, the `ACE_UUID::lock()` accessor interface and implementation was cleaned up so that it would not return a lock whose memory had been freed.

19.6.3 CIAO 0.5.4

- Adding support to manage multiple interconnected assemblies, which will involve the work through `ExternalReferenceEndPoint` idea.
- Seamless integration of swapping into the main line programming model, so that `DAnCE` can actually kickstart swapping.
- Integrated real-time event service into CIAO and `DAnCE`.
- Improved syntax error checking and reporting in the CIDL compiler.
- Add Null Component to be able to measure footprint in detail
- Added the naming service and the implementation of `createPackage` function to `RepoMan`.
- Added the code to save the state of the `RepoMan` at exit and load the state of it at start.
- Reimplemented the `findNamesByType()` and `getAllTypes()` operations of `RepoMan`, which use the newly generated `ACE_Hash_MultiMap_Manager` class and its corresponding classes.
- Added `Plan_Generator` project. This project is used to retrieve information from `Repoman` and generate/modify `DeploymentPlans` based on different demands.
- A successful static deployment of the Hello example has been tested on VxWorks 6.3

19.7 Changes in 5.5.3/1.5.3/0.5.3

19.7.1 ACE 5.5.3

- Added the `ACE_Hash_MultiMap_Manager` class and its test file.
- Changed the `ACE_Synch_Options::operator[]` method to return `bool` rather than `int`. The value returned is a yes/no indication of whether or not the specified option(s) are set in the object.

- Changed the prototype(s) for `ACE::debug ()` to return (and take) a `bool`. This is consistent with the original intent for this feature. If you have been using it like `'ACE::debug () > 0'` or `'ACE::debug (1)'`, you may have to rebuild ACE. The value of the `ACE_DEBUG` environment variable can be used to specify the initial value for `ACE::debug()`, at the process start up.
- An assembler (within a C source file) based implementation for SPARC of atomic operations suitable for use with the `ACE_Atomic_Op<ACE_Thread_Lock>` and `ACE_Atomic_Op<ACE_Thread_Mutex, unsigned long>` specializations has been added. Currently, it can only be enabled by setting the `atomic_ops_sparc` make macro to 1 when using the GNUACE build system with the Solaris SunCC compiler. It should be noted that this requires the `-xarch=v8plus` (or higher) be added to the `CFLAGS` make macro or the assembler code will not compile.
- The `ACE_Message_Queue_Ex_N<class ACE_MESSAGE_TYPE, ACE_SYNCH_DECL>` class is new, contributed by Guy Peleg <guy dot peleg at amdocs dot com>. `ACE_Message_Queue_Ex_N<class ACE_MESSAGE_TYPE, ACE_SYNCH_DECL>` is similar to `ACE_Message_Queue_Ex` in that the object queued is a template parameter. However, `ACE_Message_Queue_Ex_N` allows the enqueueing and dequeueing of multiple chained objects at once. This wasn't added to `ACE_Message_Queue_Ex` because the chained object functionality requires the `ACE_MESSAGE_TYPE` class to have a `ACE_MESSAGE_TYPE *next (void) const` method, analogous to `ACE_Message_Block::next()`, to follow the chain and this would probably break existing applications using `ACE_Message_Queue_Ex`. The `ACE_wrappers/tests/Message_Queue_Test_Ex.cpp` test has an example of how to use the new class.
- The selector and comparator function pointer arguments to `ACE_OS::scandir()` and `ACE_Dirent_Selector` are now marked as `extern "C"` to enforce their use with a C RTL function. User code that defines functions which are passed as the selector or comparator arguments which are not declared `extern "C"` may generate compile warnings. To resolve this, add `extern "C"` to the function's signature. See `ACE_wrappers/tests/Dirent_Test.cpp` for an example.
- To address a problem in the ACE string interface that prevented substring or character searches in very large strings (e.g. greater than the maximum value of an `ssize_t` type) from being correctly reported to the caller, the `find()`, `rfind()` and `strstr()` methods now return an unsigned integer (`size_t`) instead of a signed one (`ssize_t`). Affected classes include:
`* ACE_CString * ACE_WString * ACE_TString * ACE_NS_WString`
 Unless you have been explicitly using -1 instead of `npos` when comparing the return value of `find()`, `rfind()` and `strstr()`, and/or assigning the return value to `ssize_t` you should not see any difference. A new `size_type` typedef has been added to the ACE string class to aid developers. This typedef is analogous to the standard C++ `string::size_type` typedef.
 The `ACE_String_Base<>::strstr()` documentation and the default `rfind()` argument erroneously referred to -1 instead of `npos`. Those instances have been corrected.

To summarize, a "no position" condition is denoted using the `npos` constant, not `-1`. It can be referred directly by scoping it with the appropriate string class (e.g. `ACE_CString::npos`, `ACE_WString::npos`, etc).

- Changing the shared library extension for `hpux ia64` to `".so"`. On HP-UX 11i Version 1.5 the naming scheme is `lib*.sl` for PA and `lib*.so` on IPF.
- The `ACE_Refcounted_Auto_Ptr` `reset()` and `release()` methods were changed per [bugzilla 1925](#). They will both now detach from the underlying `ACE_Refcounted_Auto_Ptr_Rep` object; `reset()` will create a new one for the new pointer specified as its argument. This change may cause referenced objects to be deleted in cases where previous ACE versions would not have.
- The return type of `"ACE_Refcounted_Auto_Ptr::null (void) const"` changed from `int` to `bool`. It's possible values, `true` and `false`, have not changed.
- `TTY_IO` now accepts `"none"` as a valid parity value. Due to this change `'parityenb'` member is now deprecated and will be removed in the future. The users of `TTY_IO` class should change their code to use only `'paritymode'` member for parity control and leave `'parityenb'` unchanged (it is enabled by default in class constructor).
- Support for Intel C++ 9.1 on Windows and Linux
- VxWorks 6.3 support
- Fixed Bugzilla #2648 to make sure `ACE_Service_Object::fini()` is called iff `ACE_Service_Object::init()` succeeded, as per C++NPv2.
- Added preliminary support for Mac OS X 10.4 on Intel CPU's.
- Fixed [bugzilla 2602](#) to re-enable XML Service Configurator file support.

19.7.2 TAO 1.5.3

-

19.7.3 CIAO 0.5.3

- Added the base projects for `executionmanager_stub` and `plan_generator`.
-

19.8 Changes in 5.5.2/1.5.2/0.5.2

19.8.1 ACE 5.5.2

- Added support for: - VxWorks 6.2 for the rtp model using pthread support - OpenVMS 8.2 for Alpha
- Removed code and configurations that provided support for: - Visual C++ 6.0 and 7.0 - Chorus - pSOS - KAI C++ on all platforms
- Explicit template instantiation support has been removed. This effectively removes support for Sun Forte 6 and 7 which required explicit template instantiation to build ACE reliably.
- Added support for multiple independent Service Repositories through configuration contexts called "Gestalt". Full backwards compatibility is maintained through the existing ACE_Service_Config static methods, while direct individual repository access is enabled through instances of the new ACE_Service_Gestalt class. ACE_Service_Config has changed to a specialization of ACE_Service_Gestalt and is only responsible for the process-wide configuration.
- To support dynamically-sized ACE_Log_Record messages, the netvcs logging components now use ACE CDR encoding and transfer mechanisms inspired by the examples in Chapter 4 of the C++NPv1 book. The client and server logging daemons in ACE 5.5.2 and forward will not interoperate with those in previous ACE versions.
- Added a wrapper for the sendfile API (ACE_OS::sendfile()).
- Added support for netlink sockets on Linux.
- Added a new method, ACE_Task::last_thread(). This method returns the thread ID (ACE_thread_t) of the last thread to exit from the ACE_Task object. Users checking to see if a thread is the last one out (for example, to know when to perform cleanup operations) should compare the current thread ID to the return value from last_thread(). This is a change from the previously recommended practice (C++NPv2, page 189) of comparing the return value of thr_count() with 0.
- Changed the first argument to ACE_OS::strptime() to be 'const' which matches its usual usage in POSIX strptime(). This change allows users to pass const strings in - a common use case.
- Made part of the file support in ACE 64bit but we have some places where 32bit types are used, this could lead to some conversion warnings which will be addressed in the near future, but getting everything 64bit compliant is a lot of work.

19.8.2 TAO 1.5.2

-

19.8.3 CIAO 0.5.2

-

19.9 Changes in 5.5.1/1.5.1/0.5.1

19.9.1 ACE 5.5.1

- Added support for the `--enable-symbol-visibility` configure option to the autoconf build infrastructure instead of solely relying on feature tests to enable/disable symbol visibility support. This avoids build problems with `icc`, etc.
- Added support for the `--enable-fl-reactor` configure option to the autoconf build infrastructure to build the `ACE_FIReactor` library.
- Added support for the `--enable-qt-reactor` configure option to the autoconf build infrastructure to build the `ACE_QtReactor` library.
- Added support for the `--enable-xt-reactor` configure option to the autoconf build infrastructure to build the `ACE_XtReactor` library.
- Fixed a bug that would cause timer IDs from `ACE_Timer_Heap` to be improperly duplicated under certain conditions ([bugzilla 2447](#)).
- Fixed `ACE_SSL_Context::private_key()`, `context()`, and `dh_params()` methods to allow retrying a file load after a failed call.
- Fixed `ACE_SSL_Asynch_Stream` so it can be instantiated; also moved the declarations for `ACE_SSL_Asynch_Read_Stream_Result`, `ACE_SSL_Asynch_Write_Stream_Result`, and `ACE_SSL_Asynch_Result` classes to the `ace/SSL/SSL_Asynch_Stream.h` file so applications can see them.

19.9.2 TAO 1.5.1

-

19.9.3 CIAO 0.5.1

-

19.10 Changes in 5.5.4/1.5.4/0.5.4

19.10.1 ACE 5.5.4

-

19.10.2 TAO 1.5.4

-

19.10.3 CIAO 0.5.4

-

19.11 Changes in 5.4.10/1.4.10/0.4.10

19.11.1 ACE 5.4.10

-

19.11.2 TAO 1.4.10

-

19.11.3 CIAO 0.4.10

-

19.12 Changes in 5.4.9/1.4.9/0.4.9

19.12.1 ACE 5.4.9

-

19.12.2 TAO 1.4.9

-

19.12.3 CIAO 0.4.9

-

19.13 Changes in 5.4.8/1.4.8/0.4.8

19.13.1 ACE 5.4.8

-

19.13.2 TAO 1.4.8

-

19.13.3 CIAO 0.4.8

-

19.14 Changes in 5.4.7/1.4.7/0.4.7

19.14.1 ACE 5.4.7

-

19.14.2 TAO 1.4.7

-

19.14.3 CIAO 0.4.7

-

19.15 Changes in 5.4.6/1.4.6/0.4.6

19.15.1 ACE 5.4.6

-

19.15.2 TAO 1.4.6

-

19.15.3 CIAO 0.4.6

-

19.16 Changes in 5.4.5/1.4.5/0.4.5

19.16.1 ACE 5.4.5

-

19.16.2 TAO 1.4.5

-

19.16.3 CIAO 0.4.5

-

19.17 Changes in 5.4.4/1.4.4/0.4.4

19.17.1 ACE 5.4.4

-

19.17.2 TAO 1.4.4

-

19.17.3 CIAO 0.4.4

-

19.18 Changes in 5.4.3/1.4.3/0.4.3

19.18.1 ACE 5.4.3

-

19.18.2 TAO 1.4.3

-

19.18.3 CIAO 0.4.3

-

19.19 Changes in 5.4.2/1.4.2/0.4.2

19.19.1 ACE 5.4.2

-

19.19.2 TAO 1.4.2

-

19.19.3 CIAO 0.4.2

-

19.20 Changes in 5.4.1/1.4.1/0.4.1

19.20.1 ACE 5.4.1

-

19.20.2 TAO 1.4.1

-

19.20.3 CIAO 0.4.1

-

19.21 Changes in 5.4.0/1.4.0/0.4.0

19.21.1 ACE 5.4.0

-

19.21.2 TAO 1.4.0

-

19.21.3 CIAO 0.4.0

-

TPG Changes 20

20.1 0.11

- Added section how to setup a host build
- Added section how to build for RTEMS
- Added chapter explaining the ProtocolProperties
- Added C++ Builder with vcl information
- Added more sections with version differences

20.2 0.10

- First release of the TAO Programmers Guide