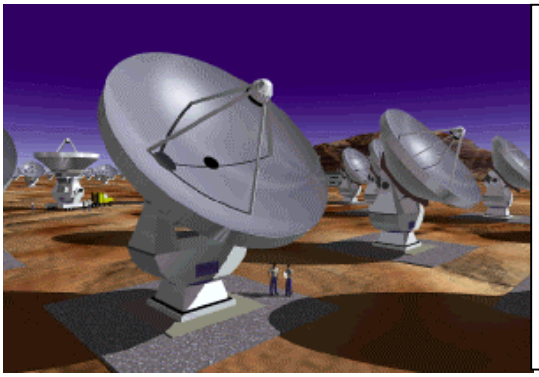

Atacama Large Millimeter Array	ALMA-NNNNN Revision: 0 2001-03-20 *ACS*

AMI/ACS Report

R.Lemke G.Chiozzi

This short report compares asynchronous invocation using AMI with asynchronous invocation using ACS callbacks.

1 AMI versus ACS callbacks

AMI is described in the following paper: "Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks", available on the ACE/TAO web site at the following URL: http://www.cs.wustl.edu/~schmidt/PDF/ami1.pdf .

For AMI two models could be used:

1.) Polling, a polling client is used via C++ method calls to find out if a request is finished and retrieves the response.

2.) Callback, a reply handler servant is used to retrieve the response from the server and notifies the client via callback

In both models the server remains the same. Only the synchronous version of the server has to be implemented.

There are the following advantages:

- it is more efficient
- a simplified asynchronous programming model could be used. On the server side only SII (static invocation interface) is used instead of DII (dynamic invocation interface).
- improved quality of service. It bounds the amount of time spend in ORB operations.

and the following disadvantages:

- IDL compiler has to generate synchronous and asynchronous (which are prefixed with sendc_) stubs
- a reply handler has to be implemented on the client side, which makes the client more complex.

In general there are 3 ways available to implement a reply handler servant:

1.) Servant-per-AMI-call strategy: a separate object is required per invocation. This has the advantage that callbacks from multiple AMI calls can be easily distinguished. On the other hand more memory is needed during multiple asynchronous calls.

2.) activation-per-AMI-call strategy: activates the same servant multiple times in the client's POA. The advantage is clearly the use of only one servant, it makes the client scalable, but is of cause more complex to program.

3.) server-differentiated-reply strategy: here the server returns an ID (e.g. a string). The advantages are the same as in 2) but more network load is involved in passing the additional information. Nevertheless this could be reduced by using Asynchronous Completion Tokens (ACTs). As the reply handler can be identified by an object reference to a remote object it should be possible to retrieve its current state (call pending, aborted, finished, ...)

As shown in the AMI example a big disadvantage of using AMI is the fact that all methods defined for the servant interface have to be implemented on the client side as well.

Another disadvantage of AMI is that it is only fully supported by TAO/ACE, all other ORBs as well as Java and Python are not supporting AMI (this might change in the future, but nobody is giving any dates).

Examples:

The complete source code for the examples comparing AMI and ACS callbacks is given in the following sections and is available on the ACS web page.

We have taken one AMI TAO example and implemented the same behavior in the easiest way with ACS callbacks.

1) AMI using activation-per-AMI-call strategy. Note: The client is sending the requests to the server which processes them in sequence. For this reasons the AMI replies are always in same sequence. Some care is needed on the server implementation to avoid waiting in one operation before a new one could be started. As can be noticed, with the example as it is now, no request is handled before a previous pending one has been completed.

2) ACS Callbacks. Note: Server and client are asynchronous. The reply sequence depends on the time spent in each operation. To have a more meaningful example, also the client here is fully asynchronous. In this case requests can be handled while previous requests are still pending.

Conclusions

Looking at the code of both examples the asynchronous server approach clearly forces the implementation of an asynchronous client as well. As a consequence it is more complex to program. But as it is a general approach it can be used on a wider range of ORBs and serves for nearly all purposes. If however AMI is needed (eg for real time performance, scalability) it can be used only with ACE/TAO.

ACS 1.0 will allow the usage of AMI, but we are convinced that ACS callbacks are more general, not more complex (actually easier with servants with a wide interface) on the client side and reasonably complex on the server side.

2 AMI example

2.1 Test.idl

```
//
// test.idl,v 1.1 1999/12/21 23:51:55 irfan Exp
//

interface test
{
  void method (in unsigned long request_number,
               out unsigned long reply_number);

  // the next method was added to demonstrate that on the client side
  // ALL methods have to be implemented, regardless if one whats to use
  // them or not!
#ifdef ADDITIONAL
  void additional_method (in  unsigned long request_number,
                          out unsigned long reply_number);
#endif
  oneway void shutdown ();
};
```

2.2 Test_i.h

```
// test_i.h,v 1.1 1999/12/21 23:51:55 irfan Exp

// ========================================================================
//
// = LIBRARY
//   TAO/examples/Buffered_AMI/
//
// = FILENAME
//   test_i.h
//
// = AUTHOR
//   Irfan Pyarali
//
// ========================================================================

#ifndef TAO_BUFFERED_AMI_TEST_I_H
#define TAO_BUFFERED_AMI_TEST_I_H

#include "testS.h"

class test_i : public POA_test
{
  // = TITLE
  //   Simple implementation.
  //
public:
  test_i (CORBA::ORB_ptr orb);
  // ctor.

  // = The test interface methods.
  void method (CORBA::ULong request_number,
               CORBA::ULong_out reply_number,
               CORBA::Environment &)
    ACE_THROW_SPEC (());
#ifdef ADDITIONAL
  void additional_method (CORBA::ULong request_number,
```

```
                CORBA::ULong_out reply_number,
                CORBA::Environment &)
    ACE_THROW_SPEC ());
#endif
  void shutdown (CORBA::Environment &)
    ACE_THROW_SPEC ());

private:
  CORBA::ORB_var orb_;
  // The ORB.
};

#endif /* TAO_BUFFERED_AMI_TEST_I_H */
```

2.3 test_i.cpp

```
// test_i.cpp,v 1.1 1999/12/21 23:51:55 irfan Exp

#include "test_i.h"

ACE_RCSID(AMI, test_i, "test_i.cpp,v 1.1 1999/12/21 23:51:55 irfan Exp")

test_i::test_i (CORBA::ORB_ptr orb)
  : orb_ (CORBA::ORB::_duplicate (orb))
{
}

void
test_i::method (CORBA::ULong request_number,
                CORBA::ULong_out reply_number,
                CORBA::Environment &)
  ACE_THROW_SPEC ())
{

  if (request_number == 1) sleep(5);

  // align to next full second
  // ACE_Time_Value txv = ACE_OS::gettimeofday ();
  // usleep(1000000-txv.usec ());
  //
  ACE_DEBUG ((LM_DEBUG,
              "server: Iteration %d @ %T\n",
              request_number));
  reply_number = request_number;
}

#ifdef additional
void
test_i::additional_method (CORBA::ULong request_number,
                CORBA::ULong_out reply_number,
                CORBA::Environment &)
  ACE_THROW_SPEC ())
{

  if (request_number == 1) sleep(5);

  // align to next full second
  // ACE_Time_Value txv = ACE_OS::gettimeofday ();
  // usleep(1000000-txv.usec ());
  //
  ACE_DEBUG ((LM_DEBUG,
              "server: Iteration %d @ %T\n",
              request_number));
  reply_number = request_number;
}
#endif

void
test_i::shutdown (CORBA::Environment &ACE_TRY_ENV)
  ACE_THROW_SPEC ())
{
  this->orb_->shutdown (0,
```

```
                        ACE_TRY_ENV);
}
```

2.4 server.cpp

```
// server.cpp,v 1.1 1999/12/21 23:51:54 irfan Exp

#include "ace/Get_Opt.h"
#include "test_i.h"

ACE_RCSID(AMI, server, "$Id: ")

const char *ior_output_file = "ior";

int
parse_args (int argc, char *argv[])
{
  ACE_Get_Opt get_opts (argc, argv, "o:");
  int c;

  while ((c = get_opts ()) != -1)
    switch (c)
      {
      case 'o':
        ior_output_file = get_opts.optarg;
        break;
      case '?':
      default:
        ACE_ERROR_RETURN ((LM_ERROR,
                           "usage:  %s "
                           "-o <iorfile>"
                           "\n",
                           argv [0]),
                          -1);
      }

  // Indicates sucessful parsing of the command line
  return 0;
}

int
main (int argc, char *argv[])
{
  ACE_TRY_NEW_ENV
    {
      CORBA::ORB_var orb =
        CORBA::ORB_init (argc,
                         argv,
                         "",
                         ACE_TRY_ENV);
      ACE_TRY_CHECK;

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

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

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

      if (parse_args (argc, argv) != 0)
        return -1;

      test_i servant (orb.in ());

      test_var server =
        servant._this (ACE_TRY_ENV);
```

```
      ACE_TRY_CHECK;

      CORBA::String_var ior =
        orb->object_to_string (server.in (),
                               ACE_TRY_ENV);
      ACE_TRY_CHECK;

      ACE_DEBUG ((LM_DEBUG, "Activated as <%s>\n", ior.in ()));

      FILE *output_file = ACE_OS::fopen (ior_output_file, "w");
      if (output_file == 0)
        ACE_ERROR_RETURN ((LM_ERROR,
                           "Cannot open output file for writing IOR: %s",
                           ior_output_file),
                          -1);
      ACE_OS::fprintf (output_file, "%s", ior.in ());
      ACE_OS::fclose (output_file);

      poa_manager->activate (ACE_TRY_ENV);
      ACE_TRY_CHECK;

      if (orb->run (ACE_TRY_ENV) == -1)
        ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "orb->run"), -1);
      ACE_TRY_CHECK;

      ACE_DEBUG ((LM_DEBUG, "event loop finished\n"));

      root_poa->destroy (1,
                         1,
                         ACE_TRY_ENV);
      ACE_TRY_CHECK;
    }
  ACE_CATCHANY
    {
      ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION,
                           "Exception caught:");
      return -1;
    }
  ACE_ENDTRY;

  return 0;
}
```

2.5 client.cpp

```
// client.cpp,v 1.5 2000/01/19 02:40:01 irfan Exp

// ========================================================================
//
// = FILENAME
//     client.cpp
//
// = DESCRIPTION
//     This is a client that uses AMI calls.
//
// = AUTHOR
//     Irfan Pyarali
//
// ========================================================================

#include "ace/Get_Opt.h"
#include "ace/Read_Buffer.h"
#include "testS.h"

ACE_RCSID(AMI, client, "client.cpp,v 1.5 2000/01/19 02:40:01 irfan Exp")

// Name of file contains ior.
static const char *IOR = "file://ior";

// Default iterations.
static CORBA::ULong iterations = 20;

// Time interval between invocation (in milli seconds).
```

```
static long interval = 1000;

// Flag indicates whether to shutdown remote server or not upon client
// shutdown.
static int shutdown_server = 0;

// AMI call or regular call.
static int invoke_ami_style = 1;

// Flag indicates that all replies have been received
static int received_all_replies = 0;

class Reply_Handler : public POA_AMI_testHandler
{
public:
  void method (CORBA::ULong reply_number,
               CORBA::Environment &)
    ACE_THROW_SPEC ((CORBA::SystemException))
    {
      ACE_DEBUG ((LM_DEBUG,
                  "client: AMI Reply %d @ %T\n",
                  reply_number));

      // Last reply flips the flag.
      if (reply_number == iterations)
        received_all_replies = 1;
    }

  void method_excep (AMI_testExceptionHolder *,
                     CORBA::Environment &ACE_TRY_ENV)
    ACE_THROW_SPEC ((CORBA::SystemException))
    {
      ACE_PRINT_EXCEPTION ((*ACE_TRY_ENV.exception ()),
                           "AMI exception caught:");
    }

#ifdef ADDITIONAL
  // NOTE: the additional_method has to be implemented if it was
  //       given in the IDL file!

  void additional_method (CORBA::ULong reply_number,
                CORBA::Environment &)
    ACE_THROW_SPEC ((CORBA::SystemException))
    {
      ACE_DEBUG ((LM_DEBUG,
                  "client: AMI Reply %d @ %T\n",
                  reply_number));

      // Last reply flips the flag.
      if (reply_number == iterations)
        received_all_replies = 1;
    }

  void additional_method_excep (AMI_testExceptionHolder *,
                       CORBA::Environment &ACE_TRY_ENV)
    ACE_THROW_SPEC ((CORBA::SystemException))
    {
      ACE_PRINT_EXCEPTION ((*ACE_TRY_ENV.exception ()),
                           "AMI exception caught:");
    }
#endif ADDITIONAL
};

static int
parse_args (int argc, char **argv)
{
  ACE_Get_Opt get_opts (argc, argv, "a:b:k:m:i:t:x");
  int c;

  while ((c = get_opts ()) != -1)
    switch (c)
      {
      case 'k':
        IOR = get_opts.optarg;
        break;
```

```
      case 'a':
        invoke_ami_style = ::atoi (get_opts.optarg);
        break;

      case 'i':
        iterations = ::atoi (get_opts.optarg);
        break;

      case 't':
        interval = ::atoi (get_opts.optarg);
        break;

      case 'x':
        shutdown_server = 1;
        break;

      case '?':
      default:
        ACE_ERROR_RETURN ((LM_ERROR,
                           "usage:  %s "
                           "-k IOR "
                           "-a invoke AMI style [0/1] "
                           "-i iterations "
                           "-t interval between calls "
                           "-x shutdown server "
                           "\n",
                           argv [0]),
                          -1);
      }

  if (IOR == 0)
    ACE_ERROR_RETURN ((LM_ERROR,
                       "Please specify the IOR for the servant\n"), -1);

  // Without AMI, replies are immediate.
  if (!invoke_ami_style)
    received_all_replies = 1;

  // Indicates successful parsing of command line.
  return 0;
}

int
main (int argc, char **argv)
{
  ACE_DECLARE_NEW_CORBA_ENV;

  ACE_TRY
    {
      // Initialize the ORB.
      CORBA::ORB_var orb =
        CORBA::ORB_init (argc,
                         argv,
                         0,
                         ACE_TRY_ENV);
      ACE_TRY_CHECK;

      // Initialize options based on command-line arguments.
      int parse_args_result = parse_args (argc, argv);
      if (parse_args_result != 0)
        return parse_args_result;

      CORBA::Object_var base =
        orb->resolve_initial_references ("RootPOA",
                                         ACE_TRY_ENV);
      ACE_TRY_CHECK;

      PortableServer::POA_var root_poa =
        PortableServer::POA::_narrow (base.in (),
                                      ACE_TRY_ENV);
      ACE_TRY_CHECK;

      // Get an object reference from the argument string.
      base = orb->string_to_object (IOR,
                                    ACE_TRY_ENV);
      ACE_TRY_CHECK;
```

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

      poa_manager->activate (ACE_TRY_ENV);
      ACE_TRY_CHECK;

      // Try to narrow the object reference to a <test> reference.
      test_var test_object = test::_narrow (base.in (),
                                            ACE_TRY_ENV);
      ACE_TRY_CHECK;

      Reply_Handler reply_handler_servant;
      AMI_testHandler_var reply_handler_object = reply_handler_servant._this
(ACE_TRY_ENV);
      ACE_TRY_CHECK;

      for (CORBA::ULong i = 1; i <= iterations; ++i)
        {
          ACE_DEBUG ((LM_DEBUG,
                      "client: Iteration %d @ %T\n",
                      i));

          if (invoke_ami_style)
            {
              // Invoke the AMI method.
              test_object->sendc_method (reply_handler_object.in (),
                                         i,
                                         ACE_TRY_ENV);
              ACE_TRY_CHECK;
            }
          else
            {
              CORBA::ULong reply_number = 0;

              // Invoke the regular method.
              test_object->method (i,
                                   reply_number,
                                   ACE_TRY_ENV);
              ACE_TRY_CHECK;

              ACE_DEBUG ((LM_DEBUG,
                          "client: Regular Reply %d @ %T\n",
                          reply_number));
            }

          // Interval between successive calls.
          ACE_Time_Value sleep_interval (0,
                                         interval * 1000);

          orb->run (sleep_interval);
        }

      // Loop until all replies have been received.
      while (!received_all_replies)
        {
          orb->perform_work ();
        }

      // Shutdown server.
      if (shutdown_server)
        {
          test_object->shutdown (ACE_TRY_ENV);
          ACE_TRY_CHECK;
        }

      root_poa->destroy (1,
                         1,
                         ACE_TRY_ENV);
      ACE_TRY_CHECK;

      // Destroy the ORB.  On some platforms, e.g., Win32, the socket
      // library is closed at the end of main().  This means that any
      // socket calls made after main() fail. Hence if we wait for
      // static destructors to flush the queues, it will be too late.
```

```
      // Therefore, we use explicit destruction here and flush the
      // queues before main() ends.
      orb->destroy (ACE_TRY_ENV);
      ACE_TRY_CHECK;
    }
  ACE_CATCHANY
    {
      ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION,
                           "Exception caught:");
      return -1;
    }
  ACE_ENDTRY;

  ACE_CHECK_RETURN (-1);

  return 0;
}
```

3 ACS example

3.1 test.idl

```
//
// test.idl,v 1.1 1999/12/21 23:51:55 irfan Exp
//

// ===================================================================
//
// = FILENAME
//     test.idl
//
// = DESCRIPTION
//     This is the test interface that uses ACS callbacks.
//     Has been written taking the AMI example and changing
//     as little as possible to use ACS callbacks instead of AMI
//     callbacks.
//
// = AUTHOR
//     G.Chiozzi, R.Lemke (Irfan Pyarali for the AMI example)
//
// ===================================================================

#include <baci.idl>

interface test
{
  void method (in unsigned long request_number,
               out unsigned long reply_number);

  // GCH/RLE
  //
  // Here we just add the definition for the asynchronous method
  // with the ACS callback parameters.
  // In AMI the asynchronous method is generated automatically by
  // the IDL compiler.
  //
  void method_async(in unsigned long request_number,
            in ESO::CBlong cb, in ESO::CBDescIn desc);

  // the next method was added to demonstrate that on the client side
  // ALL methods have to be implemented, regardless if one whats to use
  // them or not!
#ifdef ADDITIONAL
  void additional_method (in  unsigned long request_number,
                          out unsigned long reply_number);
  void additional_method_async(in unsigned long request_number,
                  in ESO::CBlong cb, in ESO::CBDescIn desc);
#endif
  oneway void shutdown ();
};
```

3.2 test_i.h

```
// test_i.h,v 1.1 1999/12/21 23:51:55 irfan Exp

// ========================================================================
//
// = LIBRARY
//   TAO/examples/Buffered_AMI/
//
// = FILENAME
//   test_i.h
//
// = AUTHOR
//   Irfan Pyarali
//
// ========================================================================

#ifndef TAO_BUFFERED_AMI_TEST_I_H
#define TAO_BUFFERED_AMI_TEST_I_H

#include "testS.h"

#if USE_ACS
#include "ace/Reactor.h"
#endif

class test_i : public POA_test
{
  // = TITLE
  //   Simple implementation.
  //
public:
  struct cbData
  {
    CORBA::ULong reply_number;
    ESO::CBlong_ptr cb;
  };

  test_i (CORBA::ORB_ptr orb);
  // ctor.

  // = The test interface methods.
  void method (CORBA::ULong request_number,
               CORBA::ULong_out reply_number,
               CORBA::Environment &)
    ACE_THROW_SPEC (());
  void method_async (CORBA::ULong request_number,
            ESO::CBlong_ptr cb,
            const ESO::CBDescIn & desc,
            CORBA::Environment &)
    ACE_THROW_SPEC (());
#ifdef ADDITIONAL
  void additional_method (CORBA::ULong request_number,
               CORBA::ULong_out reply_number,
               CORBA::Environment &)
    ACE_THROW_SPEC (());
  void additional_method_async (CORBA::ULong request_number,
            ESO::CBlong_ptr cb,
            const ESO::CBDescIn & desc,
            CORBA::Environment &)
    ACE_THROW_SPEC (());
#endif
  void shutdown (CORBA::Environment &)
    ACE_THROW_SPEC (());

private:
  static ACE_Reactor *rtc;
  static void *worker(void *arguments);

  CORBA::ORB_var orb_;
```

```
  // The ORB.
};

#endif /* TAO_BUFFERED_AMI_TEST_I_H */
```

3.3 test_i.cpp

```
// test_i.cpp,v 1.1 1999/12/21 23:51:55 irfan Exp

// ========================================================================
//
// = FILENAME
//     test_i.cpp
//
// = DESCRIPTION
//     This is the implementation of the test interface
//     that uses ACS callbacks.
//     Has been written taking the AMI example and changing
//     as little as possible to use ACS callbacks instead of AMI
//     callbacks.
//     To make the example really asynchronous, we use a reactor
//     that handles the replies to be sent to the client.
//     This is then also a simple example of the usage of the reactor.
//
// = AUTHOR
//     G.Chiozzi, R.Lemke (Irfan Pyarali for the AMI example)
//
// ========================================================================

#include "test_i.h"

/* GCH/RLE
 *
 * We include here baci and thread functionality from ACE
 */
#include "baci.h"
#include "ace/Thread.h"

/* GCH/RLE
 *
 * The ACE_Reactor is a static data member, i.e.
 * it is a singleton for the class, as well as the
 * thread where it runs.
 * Since the "real" reactor must be instantiated in the
 * thread worker function, we have here just a pointer.
 */
ACE_Reactor *test_i::rtc = NULL;

/* GCH/RLE
 *
 * This is the event handler class whose handle_timeout()
 * method is used to send replies to the client
 */
class testHandler : public ACE_Event_Handler
{
  public:
  virtual int handle_timeout(const ACE_Time_Value &tv,
                const void *arg)
  {
    ACE_TRY_NEW_ENV
      {
    Completion comp;
    CBDescOut descOut;

        test_i::cbData *data = (test_i::cbData*)arg;

    ACE_DEBUG ((LM_DEBUG,
            "server (async): Completed iteration %d @ %T\n",
            data->reply_number));

    data->cb->done(data->reply_number, comp, descOut,
              ACE_TRY_ENV);
```

```
    ACE_TRY_CHECK;
    delete data;
      }
    ACE_CATCHANY
      {
    ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION,
                "Exception caught:");
    return -1;
      }
    ACE_ENDTRY;
    return 0;
  }
};

/* GCH/RLE
 *
 * The constructor spawns the reactor's thread
 * when the first instance is created
 */
test_i::test_i (CORBA::ORB_ptr orb)
  : orb_ (CORBA::ORB::_duplicate (orb))
{
  ACE_DEBUG((LM_DEBUG,"Creating servant. Spawning reactor thread\n"));
  if(rtc==NULL)
    {
    if (ACE_Thread::spawn((ACE_THR_FUNC)worker) == -1)
      ACE_DEBUG((LM_DEBUG,"Error in spawning thread\n"));
    }
}

/* GCH/RLE
 *
 * The worker function for the thread just allows the
 * reactor to pickup and handle the queued events
 */
void*
test_i::worker(void *arguments)
{
  ACE_Reactor reactor;

  rtc = &reactor;

  ACE_DEBUG ((LM_DEBUG, "worker : Starting reactor\n"));

  while (true)
    {
    reactor.handle_events();
    }

  ACE_DEBUG ((LM_DEBUG, "worker : Close reactor\n"));

 return 0;
}

void
test_i::method (CORBA::ULong request_number,
                CORBA::ULong_out reply_number,
                CORBA::Environment &)
  ACE_THROW_SPEC (())
{

  if (request_number == 1) sleep(5);

  ACE_DEBUG ((LM_DEBUG,
              "server: Iteration %d @ %T\n",
              request_number));
  reply_number = request_number;
}

/* GCH/RLE
 *
 * This is the asynchronous method.
 * To be truly asynchronous it just installs a timer
 * that will send back the reply to the client and
 * returns immediately.
 */
```

```
void
test_i::method_async (CORBA::ULong request_number,
             ESO::CBlong_ptr cb,
             const ESO::CBDescIn & desc,
             CORBA::Environment &)
  ACE_THROW_SPEC (())
{
  int sleepTime = 1;

  if (request_number == 1) sleepTime = 5;

  ACE_DEBUG ((LM_DEBUG,
              "server (async): Iteration %d @ %T, waiting %d sec\n",
              request_number, sleepTime));

  cbData *data = new cbData;
  data->reply_number = request_number;
  data->cb = ESO::CBlong::_duplicate(cb);

  rtc->schedule_timer(new testHandler,
             (const void*)data,
             ACE_Time_Value(sleepTime));
};


#ifdef additional
void
test_i::additional_method (CORBA::ULong request_number,
                CORBA::ULong_out reply_number,
                CORBA::Environment &)
  ACE_THROW_SPEC (())
{

  if (request_number == 1) sleep(5);

  // align to next full second
  // ACE_Time_Value txv = ACE_OS::gettimeofday ();
  // usleep(1000000-txv.usec ());
  //
  ACE_DEBUG ((LM_DEBUG,
              "server: Iteration %d @ %T\n",
              request_number));
  reply_number = request_number;
}


/* GCH/RLE
 *
 * This is the asynchronous method.
 * To be truly asynchronous it just installs a timer
 * that will send back the reply to the client and
 * returns immediately.
 */
void
test_i::additional_method_async (CORBA::ULong request_number,
                                 ESO::CBlong_ptr cb,
                                 const ESO::CBDescIn & desc,
                         CORBA::Environment &)
  ACE_THROW_SPEC (())
{
  int sleepTime = 1;

  if (request_number == 1) sleepTime = 5;

  ACE_DEBUG ((LM_DEBUG,
              "server (async): Iteration %d @ %T, waiting %d sec\n",
              request_number, sleepTime));

  cbData *data = new cbData;
  data->reply_number = request_number;
  data->cb = ESO::CBlong::_duplicate(cb);

  rtc->schedule_timer(new testHandler,
             (const void*)data,
             ACE_Time_Value(sleepTime));
};
```

```
#endif

void
test_i::shutdown (CORBA::Environment &ACE_TRY_ENV)
  ACE_THROW_SPEC ())
{
  this->orb_->shutdown (0,
                        ACE_TRY_ENV);
}
```

3.4 server.cpp

same as for AMI

3.5 client.cpp

```
// client.cpp,v 1.5 2000/01/19 02:40:01 irfan Exp

// ========================================================================
//
// = FILENAME
//     client.cpp
//
// = DESCRIPTION
//     This is a client that uses ACS callbacks.
//     Has been written taking the AMI example and changing
//     as little as possible to use ACS callbacks instead of AMI
//     callbacks
//
// = AUTHOR
//     G.Chiozzi, R.Lemke (Irfan Pyarali for the AMI example)
//
// ========================================================================

#include "ace/Get_Opt.h"
#include "ace/Read_Buffer.h"
#include "testS.h"

ACE_RCSID(AMI, client, "client.cpp,v 1.5 2000/01/19 02:40:01 irfan Exp");

/* GCH/RLE
 *
 * baci include defines callback objects
 *
 */
#include <baciS.h>

// Name of file contains ior.
static const char *IOR = "file://ior";

// Default iterations.
static CORBA::ULong iterations = 20;

// Time interval between invocation (in milli seconds).
static long interval = 1000;

// Flag indicates whether to shutdown remote server or not upon client
// shutdown.
static int shutdown_server = 0;

// AMI call or regular call.
static int invoke_ami_style = 1;

// Flag indicates that all replies have been received
static int received_all_replies = 0;


/* GCH/RLE
 *
```

```
 * This callback class is used to handle the
 * callbacks from asynchronous calls
 * We are actually using only the done() method,
 * but we have to provide an implementation also for
 * the abstract working() and negotiate().
 * This callback class is anyway rather generic and can
 * be used for all remote calls.
 * On the contrary, the Reply_Handler of the AMI example
 * is specific for the test class and has to implement
 * xxx and xxx_excep methods for each single
 * method in the test interface.
 */
class MyCBlong: public virtual POA_ESO::CBlong
{
private:

public:

  void done (CORBA::Long value,
         const ESO::Completion & c,
         const ESO::CBDescOut & desc,
         CORBA::Environment &ACE_TRY_ENV =
         TAO_default_environment ())
    ACE_THROW_SPEC ((CORBA::SystemException))
  {
    static CORBA::ULong replyCounter = 0;

    ACE_DEBUG ((LM_DEBUG, "client: ACS Reply %d @ %T\n", value));

    // Last reply flips the flag.
    replyCounter++;
    if (replyCounter == iterations)
      received_all_replies = 1;
  }

  void working (CORBA::Long value,
        const ESO::Completion & c,
        const ESO::CBDescOut & desc,
        CORBA::Environment &ACE_TRY_ENV =
        TAO_default_environment ())
    ACE_THROW_SPEC ((CORBA::SystemException))
  {
    ACE_DEBUG ((LM_DEBUG, "client: ACS Working\n"));
  }

  CORBA::Boolean negotiate (ESO::TimeInterval time_to_transmit,
               const ESO::CBDescOut & desc,
               CORBA::Environment &ACE_TRY_ENV =
               TAO_default_environment ())
    ACE_THROW_SPEC ((CORBA::SystemException)) {return 1;}
};

static int
parse_args (int argc, char **argv)
{
  ACE_Get_Opt get_opts (argc, argv, "a:b:k:m:i:t:x");
  int c;

  while ((c = get_opts ()) != -1)
    switch (c)
      {
      case 'k':
        IOR = get_opts.optarg;
        break;

      case 'a':
        invoke_ami_style = ::atoi (get_opts.optarg);
        break;

      case 'i':
        iterations = ::atoi (get_opts.optarg);
        break;

      case 't':
        interval = ::atoi (get_opts.optarg);
        break;
```

```
      case 'x':
        shutdown_server = 1;
        break;

      case '?':
      default:
        ACE_ERROR_RETURN ((LM_ERROR,
                           "usage:  %s "
                           "-k IOR "
                           "-a invoke AMI style [0/1] "
                           "-i iterations "
                           "-t interval between calls "
                           "-x shutdown server "
                           "\n",
                           argv [0]),
                          -1);
      }

  if (IOR == 0)
    ACE_ERROR_RETURN ((LM_ERROR,
                       "Please specify the IOR for the servant\n"), -1);

  // Without AMI, replies are immediate.
  if (!invoke_ami_style)
    received_all_replies = 1;

  // Indicates successful parsing of command line.
  return 0;
}

int
main (int argc, char **argv)
{
  ACE_DECLARE_NEW_CORBA_ENV;

  ACE_TRY
    {
      // Initialize the ORB.
      CORBA::ORB_var orb =
        CORBA::ORB_init (argc,
                         argv,
                         0,
                         ACE_TRY_ENV);
      ACE_TRY_CHECK;

      // Initialize options based on command-line arguments.
      int parse_args_result = parse_args (argc, argv);
      if (parse_args_result != 0)
        return parse_args_result;

      CORBA::Object_var base =
        orb->resolve_initial_references ("RootPOA",
                                         ACE_TRY_ENV);
      ACE_TRY_CHECK;

      PortableServer::POA_var root_poa =
        PortableServer::POA::_narrow (base.in (),
                                      ACE_TRY_ENV);
      ACE_TRY_CHECK;

      // Get an object reference from the argument string.
      base = orb->string_to_object (IOR,
                                    ACE_TRY_ENV);
      ACE_TRY_CHECK;

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

      poa_manager->activate (ACE_TRY_ENV);
      ACE_TRY_CHECK;

      // Try to narrow the object reference to a <test> reference.
      test_var test_object = test::_narrow (base.in (),
                                            ACE_TRY_ENV);
```

```
ACE_TRY_CHECK;

/* GCH/RLE
 *
 * Instead of a Reply_Handler, here we just have to instantiate a
 * MyCBlong object, used by the servant to send back the reply.
 */
MyCBlong* mcb = new MyCBlong;
ESO::CBlong_var cb = mcb->_this();
ACE_TRY_CHECK;

ACE_TRY_CHECK;

for (CORBA::ULong i = 1; i <= iterations; ++i)
  {
    ACE_DEBUG ((LM_DEBUG,
                "client: Iteration %d @ %T\n",
                i));

    if (invoke_ami_style)
      {
    /* GCH/RLE
     *
     * The call to the remote method looks essentially the same
     * as for AMI
     */
    ESO::CBDescIn desc;
    test_object->method_async(i,
             cb.in(),
             desc,
             ACE_TRY_ENV);
    ACE_TRY_CHECK;
      }
    else
      {
        CORBA::ULong reply_number = 0;

        // Invoke the regular method.
        test_object->method (i,
                             reply_number,
                             ACE_TRY_ENV);
        ACE_TRY_CHECK;

        ACE_DEBUG ((LM_DEBUG,
                    "client: Regular Reply %d @ %T\n",
                    reply_number));
      }

    // Interval between successive calls.
    ACE_Time_Value sleep_interval (0,
                                   interval * 1000);

    orb->run (sleep_interval);
  }

// Loop until all replies have been received.
while (!received_all_replies)
  {
    orb->perform_work ();
  }

// Shutdown server.
if (shutdown_server)
  {
    test_object->shutdown (ACE_TRY_ENV);
    ACE_TRY_CHECK;
  }

root_poa->destroy (1,
                   1,
                   ACE_TRY_ENV);
ACE_TRY_CHECK;

// Destroy the ORB.  On some platforms, e.g., Win32, the socket
// library is closed at the end of main().  This means that any
// socket calls made after main() fail. Hence if we wait for
```

```
      // static destructors to flush the queues, it will be too late.
      // Therefore, we use explicit destruction here and flush the
      // queues before main() ends.
      orb->destroy (ACE_TRY_ENV);
      ACE_TRY_CHECK;
    }
  ACE_CATCHANY
    {
      ACE_PRINT_EXCEPTION (ACE_ANY_EXCEPTION,
                           "Exception caught:");
      return -1;
    }
  ACE_ENDTRY;

  ACE_CHECK_RETURN (-1);

  return 0;
}
```