Write a server according to the interface given in assignment 2. The server should register at the naming service as <id> where <id> stands for the identifier of your UNIX account. Modify the client to look up the server using the naming service, too. The client itself ought to invoke the `reverse` *operation.*

Like assignment 2 which I already handed in, this time I use the Mico 2.3.9 environment, too. Even though the IDL description remained the same, both server and client had to be modified heavily.

```
module Aufgabe2{
  interface Server{
    string reverse(in string message);
  };
};
```

The CORBA Naming Service seems to fit best to the POA based approach. Therefore, I had to rerun the IDL compiler which in turn generated slightly different classes. The abstract class named `Server` – residing inside the namespace `POA_Aufgabe2` – serves as a base class for my implementation `Server_Impl` and provides the operation `reverse`.

The main task I had to fulfill was to replace the code that connects server and client. In the previous assignment I created a shared file storing an unique IOR string. This time the CORBA Naming Service comes into play:

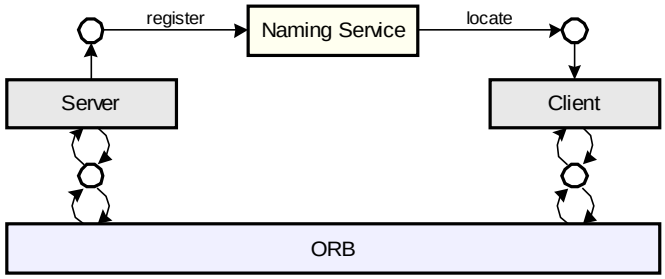

Figure 1: Block diagram

Obviously, the naming service needs to bind to an address/port that is known in advance. Due to the web servers I run on my machine, I decided to reserve port 8000 for the CORBA naming service. Setting up the daemon turns out to be quite simple:

```
nsd -ORBIIOPAddr inet:localhost:8000
```

Now that the naming service works properly, both the server and the client can connect to it. To provide a very flexible solution, my programs read the naming service's address from their command line.

```
Server -ORBInitRef NameService=corbaloc::localhost:8000/NameService
Client -ORBInitRef NameService=corbaloc::localhost:8000/NameService
```

Unfortunately, typing in long parameter lists is not friendly to the user, not to mention the developer. You might call the batch files run_nameserver.bat, run_server.bat and run_client.bat in order to get the same effect.

Resolving the naming service is a matter of only a few lines. At first, you have to get an initial reference – that is done by, suprise !, the ORB's method `resolve_initial_references("NameService")`. The next step casts the object we got to a naming context. Note that all these operations may fail or return invalid references. In consequence, performing lots of validations show up in my code:

```
////////////////////////////////////
// Connect to Naming Service
cout << "Resolve Naming Service ..." << endl;
// resolve naming service
```

```
CORBA::Object_var namingService =
  orb->resolve_initial_references("NameService");
// not found ?
CorbaAssert(namingService, "Couldn't locate the Naming Service.");
// get the naming context
CosNaming::NamingContext_var namingContext =
  CosNaming::NamingContext::_narrow(namingService);
// not found ?
CorbaAssert(namingContext, "Couldn't locate the Naming Service.");
```

To ease all these validations, I wrote a short template (can be found in Helper.h):

```
template <typename P>
void CorbaAssert(const P object, const char* msg,
                 const unsigned int code = 1)
{
    if (CORBA::is_nil(object))
        CorbaError(msg, code);
}

// displays a given error message and exits the program
void CorbaError(const char* msg, const unsigned int code = 1)
{
    std::cerr << msg << std::endl;
    exit(code);
}
```

Once a connection is established to the naming service, it is the server's responsibility to register itself. I do not know my actual HPI Unix login yet, so I will use my HPI Windows login instead: stephan.brumme. The procedure looks fairly simple:

```
// my Windows(R)(TM)(C) login name: "Stephan.Brumme"
CosNaming::Name newName;
newName.length(1);
newName[0].id   = CORBA::string_dup("Stephan.Brumme");
newName[0].kind = CORBA::string_dup("CCM");

// register server at the naming service
//namingContext->bind(newName, objectReference);
namingContext->rebind(newName, objectReference);
```

There are two ways to register a server: the most obvious one is to `bind` an object. During the develop process, especially debugging, it happens quite often that the server vanishes and is replaced by an enhanced version. The new one carries the same identifier and the same kind: it is basically the same piece with usually only minor changes. Problems arise when the new server attempts to register at the – still running – naming service: then, the naming service refuses to accept the enhanced one because of an naming conflict. My workaround calls `rebind` and thus tells the naming service to replace an existing reference if necessary.

The client does not bind, instead, it needs to resolve:

```
// create and cast the remote object
CORBA::Object_var obj = namingContext->resolve(name);
CorbaAssert(obj, "Invalid object.");
Server_var remote = Server::_narrow(obj);
CorbaAssert(remote, "Invalid object.");
```

In the code presented above, I omitted any exception handling to save space. Nevertheless, exception handling proves to be an essential part of any well-written CORBA program you should definitely use.

server.cpp

```
// ////////////////////////////////////////////////////
// Lecture on the CORBA Component Model, summer term 2003
// Assignment 3, Stephan Brumme, 702544
//
// CORBA server that provides an operation to reverse
// a string, can be reached via the Naming Service
//

#include <iostream>
using namespace std;

#include "Helper.h"

// IDL compiler generated a skeleton
#include "Aufgabe2.h"
using namespace POA_Aufgabe2;
// CORBA's naming service
#include <coss/CosNaming.h>



// the server
class Server_Impl : public virtual Server
{
public:
      // IDL: string reverse(in string message);
      char* reverse(const char* message);
};


// reverse a given message
char* Server_Impl::reverse(const char* message)
{
      // some nice info
      cout << "Invoked by client, incoming: '" << message << "'";

      // create new string
      const unsigned int length = strlen(message);
      char* result = CORBA::string_alloc(length);
      if (result == NULL)
            return result;

      // zero-terminated !
      result[length] = 0;

      // reverse the message
      for (unsigned int i = 0; i < length; i++)
            result[i] = message[length-i-1];

      // again some info
      cout << " - returning: '" << result << "'" << endl;
      // done !
      return result;
}
```

```
// here we go ...
int main(int argc, char* argv[])
{
      try
      {
            // initialize CORBA
            CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

            ////////////////////////////////////
            // POA related stuff
            cout << "Initialize POA ..." << endl;
            // first, locate the RootPOA
            CORBA::Object_var poaObject =
              orb->resolve_initial_references("RootPOA");

            // gain access to the POA server and its manager
            PortableServer::POA_var poa =
              PortableServer::POA::_narrow(poaObject);
            CorbaAssert(poa, "Failed to obtain POA server.");

            PortableServer::POAManager_var poaManager =
              poa->the_POAManager();
            CorbaAssert(poaManager, "Failed to obtain POA manager.");

            ////////////////////////////////////
            // Setup server
            cout << "Initialize server ..." << endl;
            // new server object
            Server_Impl* server = new Server_Impl;
            // generate unique server ID
            PortableServer::ObjectId_var objectId =
              poa->activate_object(server);
            // extract reference
            CORBA::Object_var objectReference =
              poa->id_to_reference(objectId.in());

            // for debug use: print IOR
            cout << orb->object_to_string(objectReference.in()) << endl;

            ////////////////////////////////////
            // Connect to Naming Service
            cout << "Resolve Naming Service ..." << endl;
            // resolve naming service
            CORBA::Object_var namingService =
              orb->resolve_initial_references("NameService");
            // not found ?
            CorbaAssert(namingService,
              "Couldn't locate the Naming Service.");

            // get the naming context
            CosNaming::NamingContext_var namingContext =
              CosNaming::NamingContext::_narrow(namingService);
            // not found ?
            CorbaAssert(namingContext,
              "Couldn't locate the Naming Service.");

            // my Windows(R)(TM)(C) login name: "Stephan.Brumme"
            CosNaming::Name newName;
            newName.length(1);
            newName[0].id   = CORBA::string_dup("Stephan.Brumme");
```

```
            newName[0].kind = CORBA::string_dup("CCM");

            // register server at the naming service
            try
            {
                  //namingContext->bind(newName, objectReference);
                  namingContext->rebind(newName, objectReference);
            }
            catch (CosNaming::NamingContext::NotFound& e)
            {
                  CorbaError("Naming context not found.");
            }
            catch (CosNaming::NamingContext::InvalidName& e)
            {
                  CorbaError("Invalid name.");
            }

            cout << "Server is ready." << endl;

            poaManager->activate();
            orb->run();

            // game over
            poa->destroy(true, true);
            delete server;

            return 0;
      }
      catch (CORBA::Exception& e)
      {
            CorbaError("Corba failed to run properly.");
      }
}
```

client.cpp

```
// ////////////////////////////////////////////////////
// Lecture on the CORBA Component Model, summer term 2003
// Assignment 3, Stephan Brumme, 702544
//
// CORBA client that calls a remote "reverse" operation
// using the naming service
//

// save and display IOR
#include <iostream>
using namespace std;

#include "Helper.h"

// IDL compiler generated a stub
#include "Aufgabe2.h"
using namespace Aufgabe2;
// CORBA's naming service
#include <coss/CosNaming.h>


int main(int argc, char* argv[])
{
      try
      {
            // initialize CORBA
            CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

            ////////////////////////////////////
            // Connect to Naming Service
            cout << "Resolve Naming Service ..." << endl;
            // resolve naming service
            CORBA::Object_var namingService =
              orb->resolve_initial_references("NameService");
            // not found ?
            CorbaAssert(namingService,
              "Couldn't locate the Naming Service.");

            // get the naming context
            CosNaming::NamingContext_var namingContext =
              CosNaming::NamingContext::_narrow(namingService);
            // not found ?
            CorbaAssert(namingContext,
              "Couldn't locate the Naming Service.");

            // my Windows(R)(TM)(C) login name: "Stephan.Brumme"
            CosNaming::Name name;
            name.length(1);
            name[0].id   = CORBA::string_dup("Stephan.Brumme");
            name[0].kind = CORBA::string_dup("CCM");


            // create and cast the remote object
            CORBA::Object_var obj = namingContext->resolve(name);
            CorbaAssert(obj, "Invalid object.");
            Server_var remote = Server::_narrow(obj);
            CorbaAssert(remote, "Invalid object.");
```

```
            // invoke the "reverse" operation
            std::cout << remote->reverse("Corba Component Model") <<
              std::endl;
      }
      catch (CORBA::Exception& e)
      {
            CorbaError("Corba failed to run properly.");
      }

      return 0;
}
```

helper.h

```
// ////////////////////////////////////////////////////
// Lecture on the CORBA Component Model, summer term 2003
// Assignment 3+, Stephan Brumme, 702544
//
// Some generic helper functions
//


#ifndef _HELPER_H_INCLUDED
#define _HELPER_H_INCLUDED


#include <iostream>


// exits the program if the given CORBA::object_var satisfies CORBA::is_nil
template <typename P>
void CorbaAssert(const P object, const char* msg, const unsigned int code =
1)
{
      if (CORBA::is_nil(object))
            CorbaError(msg, code);
}

// displays a given error message and exits the program
void CorbaError(const char* msg, const unsigned int code = 1)
{
      std::cerr << msg << std::endl;
      exit(code);
}


#endif // _HELPER_H_INCLUDED
```