Second assignment CORBA

Send the finished task by e-mail to your teaching assistant. Deadlines for the submission are on the web page of the course: http://d3s.mff.cuni.cz/teaching/middleware/.

The goal of the assignment is to obtain basic experience with mapping of CORBA IDL data types and interfaces. The interface describes a simple state server, which implements several trivial methods. Rather than implementing a specific algorithm, the client is supposed to call the interface methods in a given order, provide correct values to input parameters and display values of output parameters.

Prerequisites

The chosen CORBA middleware implementation is omniORB latest version (precompiled binaries available for older versions) for the C++ language, or TAOX11 for C++11 and later. The following knowledge is needed for the implementation:

- The syntax and semantics of the CORBA IDL language.
- The mapping of CORBA IDL to the C++ language.
- Standard functionality provided by the CORBA middleware and POA methods.
- Using CORBA middleware to implement a client side of a system in C++.
- Using CORBA middleware to implement a server side of a system in C++.

An example CORBA client and server in C++ is provided.

Assignment details

`http://d3s.mff.cuni.cz/teaching/middleware/files/as2.zip`

The IOR of a running CORBA server will be provided on the web page. The server implements the interface from the provided `master.idl` file.

Your tasks are:

1. The provided IOR references a remote object implementing the `server_i` interface with the
   ```
   connect(inout string<8> peer, inout long long key)
   ```
   method. For a simple connection test it also implements the
   ```
   short ping(in shortval)
   ```
   method, which returns the input value. Choose an unique peer string such as your name or SIS login (preferred) that fits in the character limit.

   For a successful call of the `connect()` method, the right value of `key` is needed. Using a wrong key results in a `connection_e` exception, whose attribute `cause` contains an error message including the right value of `key`. The value does not have to be parsed in the code, it is sufficient to print it and exit, then run the client again with the code as a command-line argument. It is fixed for your given `peer` name.

   A successful call returns an object of the `instance_i` interface and modifies the values of the inout parameters `peer` and `key`, which will be needed later.

2. Before subsequent calls to the instance object, its attributes `idle` and `ready` need to have a value of `true`. The `ready` attribute can be set directly, the `idle` attribute is read-only and you have to poll it in a cycle until it has the true value. You should suspend the process between the iterations of the cycle (e.g. with a `sleep(1)` call).

Not fulfilling these conditions will cause the calls to other methods of the instance to throw a `protocol_e` exception. The conditions have to be fulfilled just once, the attributes will no longer change and need not be checked anymore.

3. Now call the instance method
   ```
   get_status(in string s_key, inout count_t cnt,
   out octet_sequence_t status)
   ```
   The `s_key` parameter should be set to the string that the `connect()` method returned in the `peer` parameter as part of the first objective. The union parameter `cnt` should be initialized as a `long long` type with the value that the `connect()` method returned in the `key` parameter as part of the first objective. Wrong parameter values will result in a `protocol_e` exception.

   The method will return a sequence of octets in the out parameter `status` and set the `cnt` parameter randomly as either a `short` or `long`, with a value that will be used to index the octet sequence.

4. Call the instance method
   ```
   request(in request_t req)
   ```
   where the `req` structure should have the `index` field set to the value of `cnt` and the `data` field should contain the value of the octet with index `cnt` in the sequence. Wrong parameter values will result in a `protocol_e` exception.

   Print out the return value of the `request()` method.

5. Call the instance method
   ```
   disconnect()
   ```
   before exiting from the client.

6. Based on the interface description from the `master.idl` file, implement the server side of the interface in addition to the client side. Use the basic assignment description to implement the behavior of the server side. In case of ambiguous description, choose an appropriate alternative (which does not violate the interface, common practices etc.) and concisely document your decision. Do not forget that with the default POA policy, methods of `server_i` are executed in multiple threads and thus proper synchronization is required.

   You should use the C++ language and omniORB/taox11 to implement the server. Using different implementation and/or language is possible only after previous agreement. The client should stay in C++ to demonstrate interoperability.