

YAMI Definition Language

Copyright © 2001-2008 Maciej Sobczak

Contents

1	Introduction	2
1.1	Static interfaces	2
1.2	Description language	2
1.3	Interface compilation	3
1.4	Stubs and skeletons	3
2	YAMI Definition Language	4
2.1	Grammar	4
2.2	Examples	5
3	YDL Compiler	6
3.1	Options	7
4	C++ support	7
4.1	Type bindings	10
4.1.1	Input parameters	10
4.1.2	Output parameters	10
4.2	Exceptions	11
4.3	Example	12

1 Introduction

The *YAMI Definition Language* is a high-level layer built on top of the existing *YAMI* libraries. Its purpose is to provide static interfaces and type-safety for use in communication. For those already acquainted with systems like CORBA or DCOM, *YDL* is the counterpart of the Interface Definition Language.

The typical use of *YDL* can be expressed with the following steps:

1. Design your system in terms of static interfaces.
2. Define all interfaces in specific declarative language.
3. Compile the definitions – the result is usually some code in your chosen language for use in both clients and servers.
4. Use the code generated from interface descriptions in communication.

Let's tackle all of these points.

1.1 Static interfaces

YAMI is a dynamic infrastructure. It means that there is no compile-time nor run-time constraint on the format of messages exchanged between distributed objects. It is possible to send messages with arbitrary names or with arbitrary parameters during the same session. In some situations, the dynamic nature of *YAMI* can lead to more expressive designs. It is also necessary in those parts of the distributed system, which are developed without the prior knowledge of the possible messages sent and received.

It is not the whole story, though.

There are systems which would greatly benefit from the static definitions of interfaces, to which all communicating parties comply. If the designer of the system is able to express the communication in terms of the finite number of interfaces, with well-defined messages (including number and types of parameters), the benefit can be less error handling and cleaner code. There is no way to send or receive malformed message, so it is easier to manage the whole communication. In fact, in most languages it is possible to develop messaging infrastructure that seamlessly integrates with the rest of the language, so that the remote message looks in the code exactly like local function or method call. This is what IDL or *YDL* is all about.

1.2 Description language

In order to describe the interfaces for communication in the system, some separate language is needed to formalize the description. Well, it does not need to be separate syntactically – for example, the interface definitions for Java Remote Method Invocations are expressed in Java. In CORBA, the IDL has been

greatly influenced by the C++ syntax. In those infrastructures the interface description language allows to define not only messages but also data types that will be used for parameters in messages. *YDL* is much simpler – it allows to define the names of the messages and the sets of parameters (both for message sending and for replying), where only *YAMI* Level 2 types can be used (this, of course, applies to the current version of *YDL*). With this simplification in mind, it was easier to define the description language from scratch than to build it on some foundation taken from another language.

As a simple example, consider the following interface of the calculator server:

```
calculator
{
    add < (int a, int b) > (int c).
    sub < (int a, int b) > (int c).
    mul < (int a, int b) > (int c).
    div < (int a, int b) > (int c).
}.
```

The code above defines the `calculator` interface with 4 messages: `add`, `sub`, `mul` and `div`. Each message carries two integer parameters and the server replies with one integer value.

1.3 Interface compilation

The interface description needs to be processed in some way before it can be used further in the development. In *YDL*, the interface descriptions are compiled by a separate tool, `yd1`, which generates message scaffolding for chosen language. The source code files generated by `yd1` are then used further in the development. The exact number and format of these generated files depend on the language chosen for development. Note that files for different programming languages can be generated from single interface description. In other words, the interface description can be used for development in any language (or even in *many* languages) that is supported by *YAMI* and by `yd1` compiler. At the moment, only C++ is supported by the `yd1` compiler, but this will for sure change in the future.

1.4 Stubs and skeletons

The `yd1` compiler generates files used in both clients and servers in the distributed system. The objective is to provide the regular function or method call syntax for remote message invocations. In order to achieve this, the generated code needs to disguise the remote message as a local call. For example, in C++, the following code:

```
calc.add(a, b, c);
```

looks like regular invocation of method `add` on the object `calc`, but the client of this method does *not* know what actually happens – there can be a message sent to remote server and reply sent back. In order to achieve this integration with the language, the following pieces of software need to be provided:

- The client stub – it is installed at the client side and looks like the local method. Its responsibility is to prepare the message, sent it, wait for response and unpack the return parameters.
- The server skeleton – it is installed at the server side and looks like the scaffolding for the local method (in the case of C++, it is a base class that needs to be derived from in order to provide the actual implementation). Its responsibility is to retrieve the message, unpack the parameters, call the local method that will do the job, pack the return parameters and send the reply back to client.

Both stubs and skeletons are generated by the `ydl` compiler.

2 YAMI Definition Language

This section describes the language used to define interfaces and messages for distributed communication.

2.1 Grammar

The *YDL* can be defined by the following grammar:

```

ydl_desc      = interfaces;
interfaces    = {interface_desc}, '.';
interface_desc = interface_name, open_bracket, message_list,
                  close_bracket;
message_list  = {message};
message       = message_name, [input_params], [output_params |
                  oneway_flag], end;
input_params  = input_keyword, open_bracket, params_list,
                  close_bracket;
output_params = output_keyword, open_bracket, params_list,
                  close_bracket;
params_list   = parameter, [',', params_list];
parameter     = parameter_type, parameter_name;
interface_name = name;
message_name  = name;
parameter_name = name;
parameter_type = 'string' | 'wstring' | 'int' | 'double' |
                  'byte' | 'binary';
name          = _valid_identifer_in_target_language_;
oneway_flag   = 'oneway';

```

```

input_keyword = '<' | '<<' | 'in';
output_keyword = '>' | '>>' | 'out';
open_bracket = '(' | '{' | '[' | 'begin';
close_bracket = ')' | '}' | ']' | 'end';
end = '.';

comment = comment_begin, _any_sequence_until_EOL_;
comment_begin = '//' | ';' | '#';

```

The `comment` can start anywhere in the *YDL* description and continues to the end of the line.

The `_valid_identifer_in_target_language_` is a sequence of characters that can represent a valid identifier in the language to which the *YDL* description will be compiled.

Note also that the names used in the description file will be used in the stubs and skeletons as identifiers of related language constructs. It may happen that the names used in the *YDL* description will collide with the target language or with other names used internally by the generated code. For example, do not expect reasonable results if you use names like `class` or `void` with C++ as the target language.

2.2 Examples

Note, that the grammar allows to use many different keywords with the same meaning. For example, there are four opening brackets – they can be mixed even in the same description file. Note also that the *YDL* is not line-oriented and there can be many white spaces (and of different types) between two consecutive tokens, so that the layout of the description is not constrained.

The following examples use the style that is just recommended, but not mandatory.

Example 1.

```

myserver
{
    dothis.
    dothat oneway.
}
.

```

This is the description of the `myserver` interface with two messages. The messages do not carry any parameters, neither input nor output. However, in case of `dothis` message, the client will wait for the confirmation from the server, whereas the `dothat` message is of “send and forget” type – the client will not wait for any confirmation.

Example 2.

```
myserver
{
    print < (string msg).
}
.
```

This describes the `myserver` interface with only one message. The message accepts one input parameter of type `string` and the client will wait for the confirmation from the server.

Example 3.

```
clock
{
    gettime > (string time).
}
.
```

This describes the `clock` interface with one `gettime` message. The message does not accept any input parameters, but returns with one parameter of type `string`.

Example 4.

```
calculator
{
    add < (int a, int b) > (int c).
    sub < (int a, int b) > (int c).
    mul < (int a, int b) > (int c).
    div < (int a, int b) > (int c).
}

admin
{
    shutdown oneway.
}
.
```

This is one file describing two interfaces. The `calculator` interface is a four-operation “number-cruncher”. Each of its messages carries two input parameters of type `int` and returns with one `int` result. The second `admin` interface defines only one message, which does not carry any additional data and the client will not wait for response.

3 YDL Compiler

The `ydl` is a command-line tool that is added to the *YAMI* distribution. It allows to compile the *YDL* descriptions into stubs and skeletons for some target

language. It can be used in make-files or in integrated development environments.

The example usage of the `ydl` tool can look like:

```
$ ydl -language cpp myinterfaces.ydl
```

Note that the `ydl` tool can be used not only to process existing files (many files can be provided for single run), but can also accept descriptions from its standard input.

3.1 Options

- language** This option selects one of the supported languages. At the moment only C++ is supported with the `cpp` value. This option has no default value.
- name** This option defines the core name of the generated files, if the standard input is used to feed the *YDL* descriptions. It is ignored if the existing files are provided.
- hsuffix** (for C++) Defines the suffix for generated header files. The default value is `h`.
- cppsuffix** (for C++) Defines the suffix for generated source files. The default value is `cc`.
- namespace** (for C++) Defines the namespace where all the generated definitions will be placed. By default, no namespace is used and the code is generated at the global scope.
- stringin** (for C++) Defines the way of passing input string parameters in generated methods. The possible values are: `string` (default) and `ptr`.
- stringout** (for C++) Defines the way of passing output string parameters in generated methods. The possible values are: `string` (default), `malloc` and `new`.
- binin** (for C++) Defines the way of passing input binary parameters in generated methods. The possible values are: `vector` (default) and `ptr`.
- binout** (for C++) Defines the way of passing output string parameters in generated methods. The possible values are: `vector` (default), `malloc` and `new`.

4 C++ support

The C++ language is the first language that is supported by the `ydl` tool. It is defined in terms of rules of transformations from *YDL* descriptions to C++ code.

There are always four files generated from the *YDL* description (let's suppose that the `system.ydl` file was taken for processing):

- `system_client.h` header (the suffix can be changed), which is supposed to be `#included` by the client code. It contains the declarations of the stubs.
- `system_client.cpp` source (the suffix can be changed), which is supposed to be compiled and linked with the client code. It contains the definitions of the stubs.
- `system_server.h` header (the suffix can be changed), which is supposed to be `#included` by the server code. It contains the declarations of the skeletons.
- `system_server.cpp` source (the suffix can be changed), which is supposed to be compiled and linked with the server code. It contains the definitions of the skeletons.

Every interface is mapped to one class, in both client and server. At the client side, the class represents the group of messages that can be sent – every message is a public method of this class. Moreover, the class generated for the client contains also methods that allow to use the stub with existing *Agent*. At the server side, the generated class contains the code necessary for dispatching and error handling and the declarations of pure virtual methods that the server code is supposed to override in order to provide the actual implementation.

As a short example, consider the calculator interface:

```
calculator
{
    add < (int a, int b) > (int c).
    sub < (int a, int b) > (int c).
    mul < (int a, int b) > (int c).
    div < (int a, int b) > (int c).
}
.
```

After compiling this *YDL* description, the class generated for use by the client is:

```
class calculator
{
public:
    // constructors

    calculator(::YAMI::Agent &agent,
               const std::string &domainname,
               const std::string &objectname);
    calculator(::YAMI::Agent &agent,
```

```

        const char *domainname,
        const char *objectname);

// rebind functionality

void rebind(::YAMI::Agent &agent,
            const std::string &domainname,
            const std::string &objectname);
void rebind(::YAMI::Agent &agent,
            const char *domainname,
            const char *objectname);

// timeout functionality

void setTimeOut(int timeout);

// messages

void add(int a, int b, int &c);
void sub(int a, int b, int &c);
void mul(int a, int b, int &c);
void div(int a, int b, int &c);

private:
    // ...
};

```

(the code has been slightly reformatted)

The constructors and `rebind` methods allow to connect the stub to some existing *Agent* or to change the *Agent*. The most important are methods that disguise remote message exchange as local calls – they can be used by the client code exactly like any other local method.

On the server side, the same *YDL* description results in:

```

class calculator_Skel : public ::YAMI::PassiveObject
{
private:
    // messages

    virtual void add(int a, int b, int &c) = 0;
    virtual void sub(int a, int b, int &c) = 0;
    virtual void mul(int a, int b, int &c) = 0;
    virtual void div(int a, int b, int &c) = 0;

    // this method by default does nothing,
    // which results in message rejection

```

```

    // override it if you want different behavior
    virtual void unknownMessage
        (::YAMI::IncomingMsg &incomingmsg);

    // this is used for actual dispatching
    void call(::YAMI::IncomingMsg &incomingmsg);
};

```

(this code has been reformatted)

Note that this skeleton class is derived from the `PassiveObject` interface. This means that the implementations of this skeleton are supposed to be used as normal passive servants – they should be registered with the *Agent* on the server. Note also that the methods `add`, `sub`, `mul` and `div` are *pure virtual* methods – the server code has to provide the actual implementation of the methods, but with *YDL* it is as easy as writing local methods.

4.1 Type bindings

This section describes exactly how the *YAMI* types are mapped to C++ types. The parameter names in the *YDL* description are used in the generated code.

4.1.1 Input parameters

string Depending on the mode selected by the `-stringin` option: if the mode is `string`, the `const std::string &` form is used; if the mode is `ptr`, the `const char *` form is used.

wstring Depending on the mode selected by the `-stringin` option: if the mode is `string`, the `const std::wstring &` form is used; if the mode is `ptr`, the `const wchar_t *` form is used.

int The pass-by-value `int` is used.

double The pass-by-value `double` is used.

byte The pass-by-value `char` is used.

binary Depending on the mode selected by the `-binin` option: if the mode is `vector`, the `const std::vector<char> &` form is used; if the mode is `ptr`, the `const char *` form is used with additional parameter of type `size_t` – the name of this additional parameter is formed by appending `Size` to the name in the *YDL* description.

4.1.2 Output parameters

string Depending on the mode selected by the `-stringout` option: if the mode is `string`, the `std::string &` form is used; if the mode is `malloc` or `new`, the `char *&` form is used – note, that in the `malloc` and `new` modes, the

called function (whether stub in the client or implementation in the server) allocates the memory for the string and the calling side is responsible for deallocation.

wstring Similarly, but `std::wstring &` or `wchar_t *&` form is used.

int The pass-by-reference `int &` is used.

double The pass-by-reference `double &` is used.

byte The pass-by-reference `char &` is used.

binary Depending on the mode selected by the `-binout` option: if the mode is `vector`, the `std::vector<char> &` form is used; if the mode is `malloc` or `new`, the `char *&` form is used with additional `size_t &` parameter with name formed by appending `Size` to the normal name – note, that in the `malloc` and `new` modes, the called function (whether stub in the client or implementation in the server) allocates the memory for the binary data and the calling side is responsible for deallocation.

4.2 Exceptions

The remote messaging with the use of static interfaces influences the way errors are handled. In dynamic approach, the message is identified in the user code by separate object. This separate object is a handle to *Agent's* internal resources and the user code can use this handle – for example to ask for the message status. In static approach, where the remote messages are disguised as local method calls, it is not possible to ask the *Agent* for the message status, since there is no way to address the proper resource – there is no handle. Instead, the infrastructure generated by `ydl` compiler introduces exceptions as a way to communicate failures to the user:

```
class BadResponse : public std::runtime_error // ...
```

This exception is thrown at the client side when the server sends non-conforming reply. Non-conforming means the reply which number of parameters or their types are not correct with regard to the *YDL* description.

```
class TimeOut : public std::runtime_error // ...
```

This exception is thrown at the client side when the server does not reply during the timeout. The timeout can be set for each stub separately, see the `setTimeout` method in the generated stub.

```
class Reject : public std::runtime_error // ...
```

This exception is thrown at the client side when the server rejects the message. Note, that the message is rejected not only when the server explicitly marks the incoming message as rejected (which is impossible if the server uses *YDL* layer),

but also when the server chooses not to process the message in any way (also impossible with *YDL* servers) or when the server throws some exception during processing. This exception also covers the case when the message is rejected by the remote *Agent*, for example when the object is not found (normally seen as `eUnknownObj`) or when the level of the message is not correct with regard to remote *Agent* (normally seen as `eRejectByAgent`).

Note: these are *additional* exceptions. The normal C++ exceptions (in particular, `Overflow` and `OSError`) are still valid and should be properly handled.

4.3 Example

Let's take the example file `calc.ydl`:

```

1 calculator
2 {
3     add < (int a, int b) > (int c).
4     sub < (int a, int b) > (int c).
5     mul < (int a, int b) > (int c).
6     div < (int a, int b) > (int c).
7 }
8 .

```

The client code for the calculator example (this is reworked example from the C++ tutorial) can look like the code below:

```

1 #include "yami++.h"
2 #include "calc_client.h"
3 #include <iostream>
4 #include <string>
5
6 // for sscanf
7 #include <stdio.h>
8
9 using namespace std;
10 using namespace YAMI;
11
12 const char *serverhost = "127.0.0.1";
13 const int serverport = 12340;
14 const char *domainname = "someDomain";
15 const char *objectname = "calculator";
16
17 int main()
18 {
19     try
20     {
21         netInitialize();
22         {

```

```
23     Agent agent;
24     agent.domainRegister(domainname,
25         serverhost, serverport, 2);
26
27     calculator calc(agent, domainname, objectname);
28
29     while (1)
30     {
31         cout << "please enter the expression"
32             " (or \'quit\'): ";
33
34         string str;
35         getline(cin, str);
36
37         if (str == "quit")
38             break;
39
40         // note: I use sscanf here, because
41         // istringstream is not everywhere available
42
43         int val1, val2;
44         char op;
45
46         if (sscanf(str.c_str(), "%d%c%d",
47             &val1, &op, &val2) != 3)
48         {
49             cout << "wrong expression" << endl;
50             continue;
51         }
52
53         try
54         {
55             int result;
56
57             if (op == '+')
58                 calc.add(val1, val2, result);
59             else if (op == '-')
60                 calc.sub(val1, val2, result);
61             else if (op == '*')
62                 calc.mul(val1, val2, result);
63             else if (op == '/')
64                 calc.div(val1, val2, result);
65             else
66             {
67                 cout << "wrong expression" << endl;
68                 continue;
```

```

69         }
70
71         cout << "the result is " << result << endl;
72     }
73     catch (const exception &e)
74     {
75         cout << "exception: "
76             << e.what() << endl;
77     }
78 }
79 netCleanup();
80 }
81 }
82 catch (const exception &e)
83 {
84     cout << "exception: " << e.what() << endl;
85 }
86
87 return 0;
88 }

```

Note how the message is invoked. There is no `ParamSet` involved and the remote message looks like any other local method call. Note also how the calling object is attached to the existing *Agent*.

The example server code:

```

1 #include "yami++.h"
2 #include "calc_server.h"
3 #include <iostream>
4
5 using namespace std;
6 using namespace YAMI;
7
8 const int  serverport = 12340;
9 const char *objectname = "calculator";
10
11 class calculator_Impl : public calculator_Skel
12 {
13 private:
14     void add(int a, int b, int &c)
15     {
16         cout << "add " << a << " + " << b << endl;
17         c = a + b;
18     }
19
20     void sub(int a, int b, int &c)

```

```
21     {
22         cout << "sub " << a << " - " << b << endl;
23         c = a - b;
24     }
25
26     void mul(int a, int b, int &c)
27     {
28         cout << "mul " << a << " * " << b << endl;
29         c = a * b;
30     }
31
32     void div(int a, int b, int &c)
33     {
34         cout << "div " << a << " / " << b << endl;
35         if (b == 0)
36         {
37             cout << "dividing by 0 not allowed" << endl;
38             throw 0;
39         }
40
41         c = a / b;
42     }
43 };
44
45 int main()
46 {
47     try
48     {
49         netInitialize();
50         {
51             Agent agent(serverport);
52             calculator_Impl servant;
53
54             agent.objectRegister(objectname,
55                 Agent::ePassiveSingleThreaded,
56                 &servant);
57
58             cout << "waiting forever..." << endl;
59
60             sleep(0);
61         }
62         netCleanup();
63     }
64     catch (const exception &e)
65     {
66         cout << "exception: " << e.what() << endl;
```

```
67     }  
68  
69     return 0;  
70 }
```

Note how the messages are implemented. Note also that any exception that is thrown in the implementation of the message will be caught by the infrastructure used for dispatching the message. The result visible at the client side is the message rejection, which can be interpreted as a refusal to process the message. There is no additional information that can be sent back to client as a “remote exception” mechanism.