

# YAMI C++ Wrappers Tutorial

Copyright © 2001-2008 Maciej Sobczak

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Echo</b>	<b>2</b>
2.1	Easy client . . . . .	2
2.2	Easy client with data . . . . .	3
2.3	Full blown client . . . . .	4
2.4	Server, polling version . . . . .	6
2.5	Server, passive version . . . . .	7
2.6	Asynchronous vs. synchronous . . . . .	9
2.7	Error handling . . . . .	11
<b>3</b>	<b>Calculator</b>	<b>11</b>
3.1	Client . . . . .	11
3.2	Server . . . . .	13
3.3	Explicit event processing . . . . .	16

## 1 Introduction

If you already know the *YAMI Core Library* and you have read its tutorial, you will quickly notice that this text is almost exactly the same. The only substantial change is the source code. This apparent author's *laziness* is to show you that the *YAMI C++ Wrappers* are in fact very thin wrappers around the core library. Do not hesitate to run the examples from both tutorials in pairs to see that *YAMI* provides abstracts that are language-independent and that the connectivity between programs written in different languages is really achieved.

In this tutorial, there are two assumptions:

- Every server runs on the port 12340. This is arbitrary choice.
- Every client runs on the same machine as the server (this is for simplicity only – if you have access to the network, try the examples running the processes on different machines), and if it requires its own listening socket, its port is assigned by the operating system.

## 2 Echo

As a first example, you will write your own *echo* server and client. The echo server will wait for messages from clients and will print them on the screen. Apart from that, the server will also accept a special *shutdown* message. There will be no data sent back to clients.

### 2.1 Easy client

First, you will learn how to write simple client. In most distributed systems it is easier to write clients than servers.

The first client in this tutorial sends a message to the server that tells it to shut down. Here it is (file `shutdown.cpp`):

```
1 #include "yami++.h"
2 #include <iostream>
3
4 using namespace YAMI;
5
6 int main()
7 {
8     netInitialize();
9
10    easySend("127.0.0.1", 12340, 2,
11           "echo", "shutdown");
12
13    netCleanup();
```

```

14
15     return 0;
16 }

```

Yes, it is *that* easy.

The first line is necessary to use *YAMI* in C++.

The line with call to `easySend` alone is enough to send a message to the server. There are five parameters used in this function:

- "127.0.0.1" – the server's address. It is a string value and can have the form "comp.company.com" as well.
- 12340 – the server's port
- 2 – the level of the message sent. In this example it can be 1 as well (which means "only strings, please"). In most cases you want it to be Level2, unless you are sending a message to the component that understands only strings.
- "echo" – the name of the destination object. In *YAMI*, messages are sent to objects which means that there can be many destinations in the same server process.
- "shutdown" – the message name. Later you will see that this message causes the server to shut down.

Note that the `easySend` function is overloaded and that there is another version of this function that accepts additional parameter. We will use it later.

Note also the `netInitialize` and `netCleanup` functions. They are needed on MS Windows system. If you do *not* plan to write for Windows, you can safely omit them (they are empty for Unix/Linux) but keep in mind that using them will add to portability of your code.

Interesting? So let's pass some data with the message.

## 2.2 Easy client with data

In the previous section, the message had only a name. In many cases it is enough, because it allows the server to make decisions about what action to perform, but there are situations where it would be nice to pass some data together with the message. In *YAMI*, there are two levels of messages with regard to the data that can be sent:

- Level1 – this level allows only strings and wide strings. It was designed for the components written in languages that do not understand other data types.
- Level2 – this level allows strings, wide strings, integers, doubles, bytes and arbitrary binary blocks.

Both levels are accessible in *YAMI C++ Wrappers*.

There can be many values of different types that can go with the same message at once. All the values are stored in the so-called *Parameter Set*, which is basically a list of values of chosen type. Such a *Parameter Set* has to be created before the message is sent and appended to it. It can be even used many times with different messages, but we will not use this feature here.

The client that causes the server to print “Hello, YAMI!” message on the screen can look like here (file `hello.cpp`):

```

1 #include "yami++.h"
2
3 using namespace YAMI;
4
5 int main()
6 {
7     netInitialize();
8
9     ParamSet params(1);
10    params.setString(0, "Hello, YAMI!");
11
12    easySend("127.0.0.1", 12340, 2,
13           "echo", "print", params);
14
15    netCleanup();
16
17    return 0;
18 }
```

Yes, it is *that* easy.

The line 9 defines a local variable that is a *Parameter Set* object with 1 uninitialized parameter. After that, the parameter is set to be a string of value “Hello, YAMI!”. As you can see in the next line, the *Parameter Set* object is used to send the message. Note also that the message name has changed. The *Parameter Set* object is destroyed automatically.

### 2.3 Full blown client

The examples written so far have one thing in common: they send only one message to the server and quit. There is nothing that prevents you from sending millions of messages using the `easySend` function, but in terms of performance it will not be a good idea. The `easySend` function creates a *YAMI Agent* that is responsible for actually sending a message and destroys it after that. If you want to send many messages, it is a better idea to follow the scheme:

1. Create the *Agent* object.

2. Send the message with the help of the *Agent*. Repeat this step as many times as you wish.

In this section you will write the client according to this scheme. The client itself will be smarter than before, too – it will read the lines of text from its standard input and send each line as a separate message to the *echo* server. The code is below (file `printall.cpp`):

```

1 #include "yami++.h"
2 #include <iostream>
3 #include <string>
4
5 using namespace YAMI;
6 using namespace std;
7
8 int main()
9 {
10     netInitialize();
11     {
12         Agent agent;
13         agent.domainRegister("echoserver",
14             "127.0.0.1", 12340, 2);
15
16         string line;
17         while (getline(cin, line))
18         {
19             ParamSet params(1);
20             params.setString(0, line);
21
22             agent.sendOneWay("echoserver",
23                 "echo", "print", params);
24         }
25     }
26     netCleanup();
27
28     return 0;
29 }

```

The *Agent* is created with its own listening port assigned by the operating system. This *Agent* does not use this port, since no information is sent back to the client. Please read about *Policies* to learn how to create the *Agent* object without the listening socket.

After the *Agent* is created, the remote domain is registered in it, like in *White Pages Book*. The address and port of destination *Agent* (the one created by the server) is remembered under the name "echoserver" for later use (the name in the address book is arbitrary).

The program performs a loop where it reads lines of text from standard input. Each line is then sent as a parameter with the message to the server.

Note that the `sendOneWay` function allows to send the message without any response from the server. The responses will be used in later examples.

Note also the artificial scope created by the curly braces. They are to ensure that the *Agent* object will be destroyed *before* the `netCleanup` function is called (such a scope can be created in any other way valid in C++, for example by a function). As noted above, on Unix system neither the cleanup function nor the scope would be necessary.

## 2.4 Server, polling version

Finally, we will write the server.

Err... what does this *polling* mean? First, write the following code (file `servera.cpp`):

```

1 #include "yami++.h"
2 #include <iostream>
3 #include <string>
4
5 using namespace YAMI;
6 using namespace std;
7
8 int main()
9 {
10     netInitialize();
11     {
12         Agent agent(12340);
13         agent.objectRegister("echo",
14                             Agent::ePolling, NULL);
15
16         cout << "server started" << endl;
17         while (1)
18         {
19             auto_ptr<IncomingMsg> incoming
20                 (agent.getIncoming("echo", true));
21
22             string msgname(incoming->getMsgName());
23
24             if (msgname == "shutdown")
25             {
26                 cout << "received the shutdown message"
27                     << endl;
28                 break;
29             }

```

```

30         else // if (msgname == "print")
31         {
32             auto_ptr<ParamSet> params
33                 (incoming->getParameters());
34
35             string line;
36             params->getString(0, line);
37             cout << line << endl;
38         }
39     }
40 }
41 netCleanup();
42
43 return 0;
44 }

```

After the *Agent* is created, the *polling* object is registered in it as a potential target of messages. If any message comes with the object name that is equal to the name of the registered object, the *Agent* will store the message in a queue, one queue for one object. Later, the server program goes into a loop that... asks the *Agent* if there are any messages for the *echo* object and processes them. This is why the server is *polling* – it has to ask the *Agent* for every new message. Moreover, in this example, the server will block waiting for new message if the queue is empty – the last parameter to `getIncoming` method is `true`. When the message is finally retrieved, the server checks its name. If it is "shutdown", it breaks the loop and terminates. Otherwise (which means that the message name is "print"; in real system you would also take some steps if the message has unknown or unacceptable name), the server gets the *Parameter Set* from the message and prints its first parameter on the standard output.

## 2.5 Server, passive version

If there was a *polling* server, then there must be some alternative. It is called a *passive* server. Previously, the server had to ask for each incoming message. The advantage was that the server could ask in the most appropriate time if it had some other things to do. However, in some situations it is convenient if it is the *Agent* who tells the server that there is new message. In this scheme, the server itself is passive – it only waits for invocations from the *Agent*. It is important to note that the invocations are made from the separate thread that belongs to the *Agent* object. Let's see the code (file `serverb.cpp`):

```

1 #include "yami++.h"
2 #include <iostream>
3 #include <string>
4
5 using namespace YAMI;

```

```
6 using namespace std;
7
8 class Server : public PassiveObject
9 {
10 public:
11     Server(Semaphore &s) : sem_(s) {}
12
13     void call(IncomingMsg &incoming)
14     {
15         string msgname(incoming.getMsgName());
16         if (msgname == "shutdown")
17         {
18             cout << "received the shutdown message"
19                 << endl;
20             sem_.release();
21         }
22         else // if (msgname == "print")
23         {
24             auto_ptr<ParamSet> params
25                 (incoming.getParameters());
26
27             string line;
28             params->getString(0, line);
29             cout << line << endl;
30         }
31
32         incoming.eat();
33     }
34
35 private:
36     Semaphore &sem_;
37 };
38
39 int main()
40 {
41     netInitialize();
42     {
43         Semaphore sem(0);
44         Server servant(sem);
45
46         Agent agent(12340);
47         agent.objectRegister("echo",
48                             Agent::ePassiveSingleThreaded,
49                             &servant);
50
51         cout << "server started" << endl;
```

```

52
53         sem.acquire();
54
55         // the process quits this block
56         // only when the semaphore is released
57     }
58     netCleanup();
59
60     return 0;
61 }

```

The most important difference in the `main` function is that the object was registered as a `ePassiveSingleThreaded` instead of `ePolling`. Also, the code implementing the server's functionality is encapsulated in separate class. The last parameter to the `objectRegister` function is a pointer to the servant object. Its class implements the `call` function from the `PassiveObject` interface that will be called by the *Agent* when there is some new message for the given object. After registering the object, the server process suspends its execution with simple trick – acquire the semaphore that is initially set to 0. In other words, the server process is going to sleep.

Now, let's go back to the servant. The name was chosen because in other distributed infrastructures (notably CORBA) the servant is a piece of software that is responsible for processing messages. This is exactly the purpose of this object. Note that the `incoming` object is already provided. What does the `call` function do? The same as before, with few differences:

- There is no loop to break – in order to wake up the main server thread, the global semaphore is released. This will cause the main thread to resume execution, clean up and terminate.
- The incoming message is marked as *eaten*. Without this, the *Agent* would try to send back to the client some information about the outcome of the processing and in this example the client does not want to retrieve anything. To be exact, the *Agent* expects that the message will be either processed or rejected, when “processed” means replied or eaten. Since there is no reply and no rejection, we mark the message as eaten.

## 2.6 Asynchronous vs. synchronous

*YAMI* sends and processes messages asynchronously by default. This means two things:

- When you “send” the message, it is only put into the sending queue. There is a separate sender thread in the *Agent* that is responsible for actually sending messages. The advantage? You can put many messages into the queue and go back to your work without waiting for them to be physically sent.

- When the message arrives at the remote *Agent*, it is put into the queue associated with each registered object and waits there for processing. It is natural with polling servers (the messages have to wait *somewhere* until you ask for them), but is also used by default for passive servers. The advantage? The message can be quickly received and stored while you are busy with long computations or processing previous messages.

The asynchronous sending and receiving is not always good, however. Let's say that you run the command:

```
$ ./printall
```

*(I assume some Unix environment here)*

and later type some lines of text by hand. You are not really fast with typing, so the messages (each message contains one line of text) are actually sent to the server before you get your finger off the keyboard. But try this little experiment:

```
$ ./printall < printall.cpp
```

which should sent as many messages to the server as there are lines in the `printall.cpp` file. After reaching end-of-file, the process terminates... which can happen before the sender thread sends anything! Or you will see only few of the lines on the server side. There are three solutions to this problem:

- Put some delay at the end of the client process, but before the *Agent* is destroyed, so that the sender thread will have a chance to send all messages from its queue. Do not tell anybody.
- Rewrite the client and server so that the server will send back some response after each message. Wait for this response before the client sends the next message. Do not tell anybody that the excessive network traffic is your fault.
- Create the *Agent* in the client process *without* the separate sender thread. In such configuration, every message will be sent immediately, without storing in any queue – it will be synchronous. This way you will insure that each line is physically sent to the remote end (to the server) before next line is read from the standard input. Show your code to everybody.

Similar issues arise on the server side. The messages received from the network are automatically stored in the queue. This queue can get overflowed (you can increase its maximum size, but again – do not tell anybody). Similarly, the professional solution is to change the dispatching mode to synchronous by switching off dispatching threads in the *Agent*.

Look at the `printsync.cpp` and `servsync.cpp` files to see how to do it. If you compile them, you can try the test:

```
$ ./servsync
```

in one console and (probably on different computer in the network, but you should change the address used in the client source code) in the other:

```
$ ./printsync < somelongfile
```

Enjoy.

## 2.7 Error handling

Many different things can go wrong in the distributed system. Some of them will be your fault, some not. Almost every *YAMI* function can throw an exception. In the example programs presented so far there was no exception management – for simplicity and to avoid cluttering the source code. In real programs, you will for sure want to check for and manage the exceptions. The `printsync.cpp` and `servsync.cpp` files are written with simple, but consistent exception handling.

## 3 Calculator

In this second example, you will write the client-server pair where the client asks for something and the server *responds* to the messages, sending some data back to client. This is probably the most common way of developing distributed systems.

The calculator example consists of:

- The server that accepts messages: `add`, `sub`, `mul` and `div`, each with two integer parameters. Each message will be replied to, with the appropriate integer result.
- The client that sends messages asking the server to perform some computation.

I'm sure that after this example you will be able to write distributed systems of arbitrary complexity.

### 3.1 Client

Let's start with easy things. The client (file `calcclient.cpp`) looks like here:

```
1 #include "yami++.h"
2 #include <iostream>
3
4 using namespace YAMI;
5 using namespace std;
6
7 const char *serverhost = "127.0.0.1";
8 const int  serverport  = 12340;
9 const char *domainname = "someDomain";
```

```
10 const char *objectname = "calculator";
11
12 int main()
13 {
14     netInitialize();
15     {
16         Agent agent;
17         agent.domainRegister(domainname,
18                             serverhost, serverport, 2);
19
20         ParamSet paramset(2);
21
22         paramset.setInt(0, 100);
23         paramset.setInt(1, 20);
24
25         auto_ptr<Message> msg(agent.send(domainname,
26                                         objectname, "add", paramset));
27
28         msg->wait();
29
30         Message::eStatus status = msg->getStatus();
31
32         if (status == Message::eReplied)
33         {
34             auto_ptr<ParamSet> retpar(msg->getResponse());
35
36             int result = retpar->getInt(0);
37
38             cout << "the result is " << result << endl;
39         }
40         else if (status == Message::eRejected)
41         {
42             cout << "the last message was rejected"
43                 << endl;
44         }
45         else
46         {
47             cout << "no correct reply" << endl;
48         }
49     }
50     netCleanup();
51
52     return 0;
53 }
```

The main difference to the previous example is that the `send` method is used instead of `sendOneWay`. The `Message` class encapsulates the message token. It allows you to retrieve some information about the message sent to the remote objects. It also allows you to *synchronize* the client activity with the server – the `wait` function allows the client to wait for some change in the message's status.

As you can see, the message is sent with two integer parameters in the *Parameter Set* (100 and 20). The message is sent asynchronously, so that you can continue your job without waiting for response and ask for it later. In this example there is nothing interesting to do anyway, so we decide to *wait* (with the call to `wait`) until something interesting happens to the message token. The process wakes up when (for example) the response arrives. The `status` of the message is examined to find out if there is a real reply or maybe some network error or something else. If there is a reply, we just retrieve the returning *Parameter Set* and print its only (first) parameter.

The Calculator example shows also that there are two contexts where *Parameter Set* can be used:

- When the client sends a message to the server. The *Parameter Set* can be supplied at the client-side and retrieved at the server-side.
- When the server replies to the message. The server can supply a *Parameter Set* and the client can extract it after receiving the response.

These two contexts allow to send data in both directions.

Easy? So take also a look at the `calcclient2.cpp` file and play with it – it is an interactive calculator console.

There is one thing to remember, though. The `wait` function is your friend, but do not trust him. It may happen that the server crashes in the middle of the computations. Then – the client does not receive any notification and the status of the message is always `ePending`. If you call the `wait` function, you are in troubles (the other solution, sometimes reasonable, is to periodically ask the message token if the response arrived – the client can decide by himself that something went wrong after, say, 10th try). To help you with this problem, the *Agent* object provides the *waker* service. The `calcclient2.cpp` file shows how to use it.

## 3.2 Server

The server is presented below (file `calcserver.cpp`):

```

1 #include "yami++.h"
2 #include <iostream>
3
4 using namespace YAMI;
5 using namespace std;
6
```

```
7 const int serverport = 12340;
8 const char *objectname = "calculator";
9
10 class Server : public PassiveObject
11 {
12 public:
13     void call(IncomingMsg &incoming)
14     {
15         string msgname(incoming.getMsgName());
16
17         cout << "I have received the message: "
18              << msgname << endl;
19
20         if (msgname != "add" &&
21             msgname != "sub" &&
22             msgname != "mul" &&
23             msgname != "div")
24         {
25             cout << "unknown name - rejecting" << endl;
26             incoming.reject();
27             return;
28         }
29
30         auto_ptr<ParamSet> params(
31             incoming.getParameters());
32
33         int val1 = params->getInt(0);
34         int val2 = params->getInt(1);
35
36         int result;
37         if (msgname == "add")
38         {
39             result = val1 + val2;
40         }
41         else if (msgname == "sub")
42         {
43             result = val1 - val2;
44         }
45         else if (msgname == "mul")
46         {
47             result = val1 * val2;
48         }
49         else /* msgname is "div" */
50         {
51             if (val2 == 0)
52             {
```

```

53             cout << "dividing by 0 not allowed"
54                 << endl << "rejecting" << endl;
55             incoming.reject();
56             return;
57         }
58         else
59             result = val1 / val2;
60     }
61
62     ParamSet returnparams(1);
63     returnparams.setInt(0, result);
64
65     incoming.reply(returnparams);
66 }
67 };
68
69 int main()
70 {
71     cout << "starting the server" << endl;
72     netInitialize();
73     {
74         Server servant;
75         Agent agent(serverport);
76         agent.objectRegister(objectname,
77             Agent::ePassiveSingleThreaded,
78             &servant);
79
80         cout << "going to sleep..." << endl;
81         sleep(0);
82     }
83     netCleanup();
84
85     return 0;
86 }

```

This is a *passive* server, because it uses the *call-back* pattern for message dispatching. You have seen it already in the Echo example.

This server has one design quirk. I show you this, because it is sometimes used in servers working in distributed systems (I have seen it in some CORBA examples, really). Namely – after setting up everything in the `main` function, the main server thread goes to sleep... and never wakes up. The `sleep` function with 0 as a parameter sleeps *forever*. This means two things:

- There is no way to remotely shut the server down. Locally, the administrator can just kill the server process and bless the operating system that will hopefully garbage-collect all the resources used by the process.

- All the code after the sleep function is nothing but a bluff. It will never execute.

I do not advocate this style of writing servers, although, as I've pointed out, I *have* seen it. The two clean solutions that I recommend are:

- Provide additional shut-down message, like in the Echo example. Synchronize servant function with the main thread if the server is passive or just break the message loop if it is polling.
- Provide (register) additional shut-down *object* and after setting up everything in the `main` function, call the `getIncoming` function on this object instead of going to sleep. The first message that will be received for this special shut-down object will wake you up, so that you can terminate the server in a clean way.

The file `calcserver2.cpp` is the same as presented above `calcserver.cpp` but with simple exception-handling added.

### 3.3 Explicit event processing

This part of the tutorial is aimed at advanced users, who may wish to experiment a little or who find themselves in unusual situations like the necessity to write software on platforms that do not have threads (like classic Unix or some embedded systems). Basically, you will rarely (if ever) need to write your own event processing.

It is possible to compile the *YAMI* library without threading. Then, most of the library's functionality is gone and it may seem at first that it is not possible to use *YAMI* at all. But it is.

In order to correctly process connections coming from remote *Agents*, the local *Agent* performs a loop, where new connections are managed and where everything is made. In fact, every thread in the *Agent* performs its own loop, but the only really necessary is the loop in the receiver module. There are two situations where there may be no separate receiver module:

- The target platform does not support threading.
- The programmer wants to have ultimate control over *when* the event (the incoming connection) is processed.

In both situations the programmer needs to perform the loop by himself, usually in the main part of his program – this applies both to client and server. It is also possible to combine this *explicit event processing* with other loops that may appear in the crucial part of your programs, like in windows-based GUI applications.

The example programs `clientloop.cpp` and `serverloop.cpp` are similar to the examples presented so far, but their *Agents* are prepared to work without separate threads and the important event processing is triggered explicitly in

the application code. Previously, the event processing was hidden in the receiver module which was running by itself.

The interesting thing to note is that the explicit event processing results in best performance, since everything related to message processing is done in the context of a single thread and most (or all) of the synchronization overhead is avoided. Nice, isn't it?. Of course, this comes at the cost of increased complexity, as usual.

Last few comments:

- Always handle exceptions. It will considerably save your debugging time. This tutorial does not contain exception handling only for brevity.
- Read the *YAMI C++ Wrappers Reference*.

Enjoy! And, of course, tell your friends about *YAMI*.