Network Object in C++

Final Project of HonorOS

Professor David Maziere

Po-yen Huang (Dennis)
Dong-rong Wen
May 9, 2003

Table of Content

Abstract

This project tends to find a simple solution for remote object operation in C++. Most of the common object-oriented features are supported and the performance is also a main concern. Moreover, it provides the transparency and needs the least codes modification. At last, it also makes sense for the simple implementation, which is only one thousand lines of codes.

Introduction

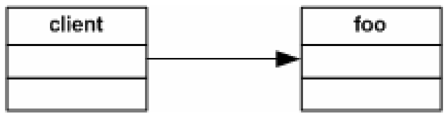
Distributing computing is getting used heavily. The power of personal computer increases dramatically, hence people prefer to use several related-slow computers than one related-fast computer. Network makes this possible and the hardware is ready for distributing computing.

Software, however, moves much slowly than hardware does. To achieve communication between two computers, software programmer often needs to deal with the low-level socket operation. The task is so lousy and error prone that keeps software from distributing computing. Remote Procedure Call (RPC), therefore, is born for this purpose. However, RPC is designed for non-object oriented environment and lose many advantages which object-oriented programming offers. Although there is some modern technique/protocol like Corba and DCOM for distributed object, they are far too complex for daily programming. Java also provides the language build-in remote invocation called Remote Method Invocation (RMI.) But it is limited only in Java and is not transparent and very slow. Here, we try to find a simple solution for C++ programmer to implement the remote object and offers the transparency and ease of use.

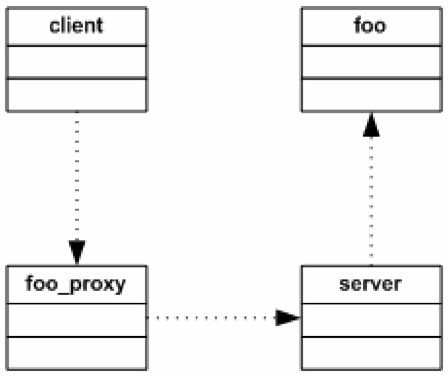
Architecture

The idea

The scenario is like this. There is a program A, which uses an object "foo." Our goal is to put program A and object "foo" running on difference memory space, and potentially difference hardware nodes.



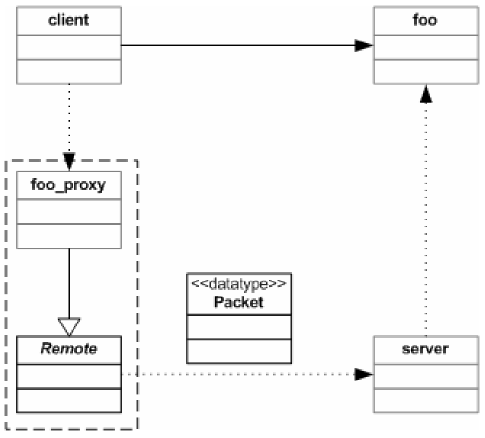
The idea is quit simple. We replace the real foo object with a "fake" foo object on the same machine on which program A runs. The fake object works like a proxy of a real foo object, let's call it foo proxy or proxy object. The proxy object implements foo's interface, namely, it has all public functions which the original foo object has. However, the proxy object does not do exactly what the original object does. Instead, it sends a request to a remote machine B when a call occurs and gets the reply from the machine B. All functions will be executed on machine B. Moreover, there is a server program running on machine B receiving all requests from the proxy object. The server then dispatches the request to appropriate object (in this case, foo object) and actually executes the function.



More details – helper classes

The proxy object plays as a foo object to the client program and deals with all the communication stuff in it. In order to make the implementation as transparent as possible, the proxy object would be better to be produced by machine. The ideal goal is need no programmer taking care of this proxy object. To keep the proxy object simple and clean and hide all the dirty trick from programmer, we pull all the real codes from the proxy object to an object called Remote. The proxy object gets the communication ability by inheriting from Remote object. With this design, the proxy object is so trivial that there will be no problem for a simple parser to produce it.

Moreover, there must be some protocol between the proxy object and the server. The proxy object and the server talk in plain TCP/IP channel. To define a request format and make it easy to use, we also made a Packet object between them. The Packet object works as a transporter between the proxy object and the server. All information is packed into the Packet object and travel between two machines.

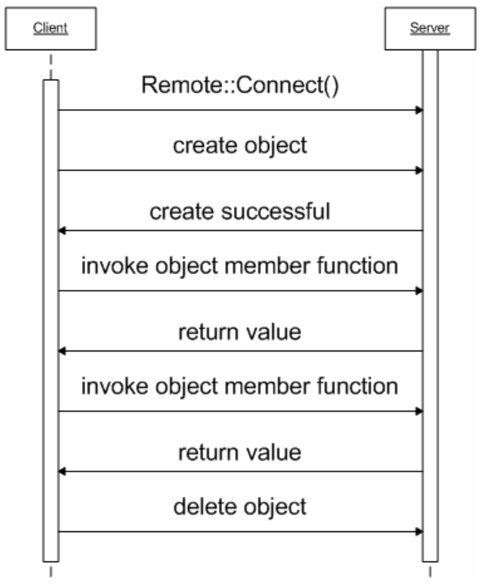


Although we didn't implement the proxy parser, it could be fairly easy to build. The sample proxy codes are listed in appendix.

Dialogue between proxy and server

The dialogue is easy to be understood. The proxy first makes a connection to a server. After that, the proxy may request the server for three different actions, create an object, destroy an object, or invoke a member function.

When creating an object, the server will new an object and send back unique object identification (object id) to the proxy. The proxy may also ask the server to destroy a specific object by its object id. More important, and most of time, the proxy will ask the server to execute a specific member function by the object id and a function signature. The arguments will also be sent over the wire as well as the return value. The dialogue is show below.



Marshalling

When a member function called in program A (client), the proxy has the responsibility to issue a request over the wire to the server. Namely, the proxy should be able to transform the call into something which can be sent over wire. We called this behavior "marshalling."

As the earlier paragraph mentioned, the proxy and the server talk in a special format called Packet object. The proxy also inherits from the Remote object. In a body of a function call of a proxy object, it simply encapsulates the arguments and call the Remote::invoke() function to send the request out. The real marshalling happens in the Remote::invoke() function. The Remote::invoke() calls the marshalling interface of a Packet object and provides the call information, including the object id, the object name, the function signature, and the arguments. The Packet object then “flattens” all the information and transmitted it over wire.

Picking the arguments

Okay, it’s easy to say “flatten all the arguments into byte stream”, but how? Objects are instances live in a specific memory space. It’s not so trivial to move an instance from one memory space to another. Here, we have two choices.

```
    Approach one: pickling all objects includes primitive type
    Approach two: all objects implement serialize interface including
primitive type
```

For the first approach, we simply save the memory content owned by an object or a primitive type. This approach is quite trivial and has the benefit of easy implementation and better performance. The main drawback is that it can deal only with simple object. It can not deal with any object which includes a pointer to another variable or inherits from other objects.

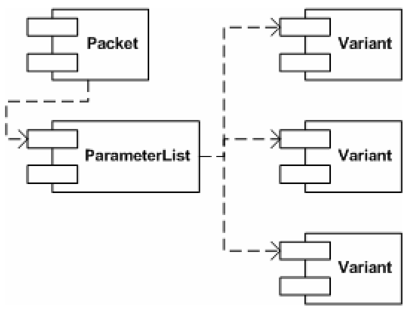
The second approach provides the ability to “flatten” all kinds of object. But this approach requires every object to implement a serialize interface and we also need the serialize mechanism for primitive type. Although this approach is universal for all objects, it lacks of easy implementation and has performance impact from the serialize mechansim.

In this project, the first approach is more suitable. We sacrifice the full-feature approach and adopt the second one for simplicity and performance.

Variant object and ParameterList object

The Variant object provides a unique “appearance” of an argument. The constructor of a Variant object takes one parameters and save a pointer points to the memory address of that parameter. The parameter could be any type by the use of C++ template. It also uses the conversion function/operator of C++ to convert the content from byte-stream into its original type. Of course, the type name of the parameter is retrieved through the run-time type identification library (RTTI) of C++ and saved in the Variant object.

We also implement a ParameterList object which contains several Variant pointers. The ParameterList does nothing but aggregate several Variant objects together. It provides an interface to easily access all the Variant objects in a function call. The relationship between Packet object, Variant object and ParameterList object is showed below. After that, the linear format of the three object are showed.



Packet

action	obj_id	len(obj_name)	obj_name	len(func_name)	<func_name>	nParam	< len(parameter_list)	parameter_list >

ParameterList

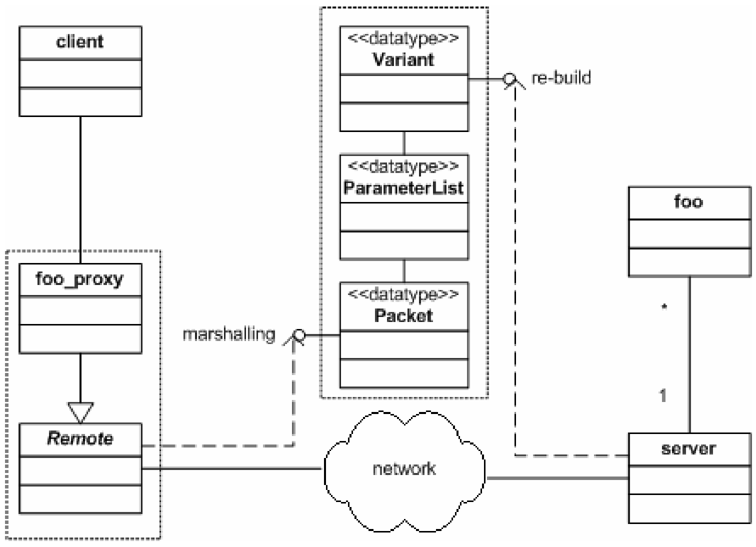
<Variant>	<Variant>	<Variant>	• • • •

Variant

len(real_value)	real_value	len(type)	type

There is one more importance thing need to be mentioned. All the three objects do NOT actually contain a copy of the argument. Instead, they only save pointers points to the argument. All the pickling operation happens when Remote::invoke() calls the marshalling interface of Packet object. There is no redundant data copy occurs. Namely, this one-copy mechanism guarantees the better performance of the pickling.

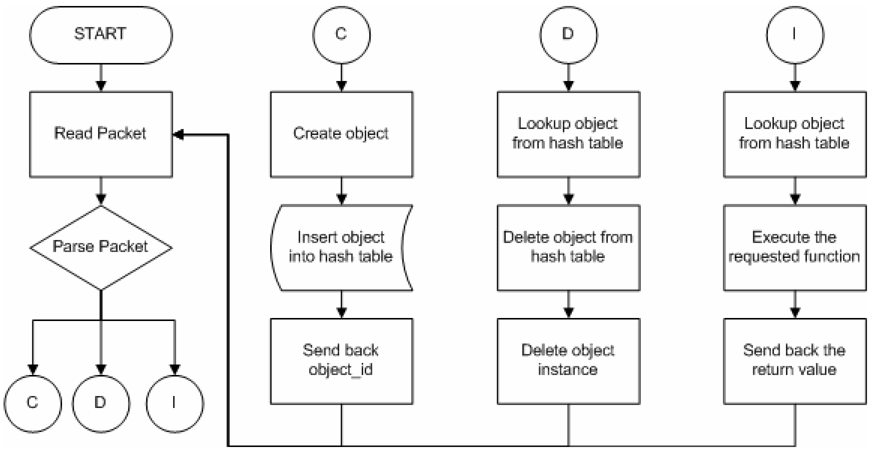
The big picture



Here we have a system-wide view of our architecture. The client calls the proxy object instead of a real foo object. The proxy object inherits from the Remote object and gains the ability of talk to the server. When an actual call happens, Remote object calls the marshalling interface of Packet and encapsulate all call information into the Packet, which contains ParameterList object and Variant object. After the Packet travels to the server, the server re-build the arguments from the Variant and execute the requested function by its object id, object name, and function signature and pass the arguments in. The server also returns the result to the proxy in the same Packet format.

Server Behavior

The server accepts three actions from the proxy, create object, destroy object and invoke a member function. The server keeps a hash table for mapping the unique object id and the pointer points to the real object instance. When creating an object, the server also inserts the object into the hash table and returns the automatically generated unique id to the proxy. When destroying an object, the server pulls out the object from the hash table and deletes the object. If invoking the function is the action, the server lookup the object from the hash table and execute the function by the function signature with the correct arguments. This behavior is showed below.


START
Read Packet
Parse Packet
C
D
I
Create object
Insert object into hash table
Send back object_id
Lookup object from hash table
Delete object from hash table
Delete object instance
Lookup object from hash table
Execute the requested function
Send back the return value

Evaluation

In this project, we used three different ways to evaluate our implementation of network object in C++. These three evaluations are:

1. Call benchmark,
2. Throughput benchmark– wired version
3. Throughput benchmark – local version

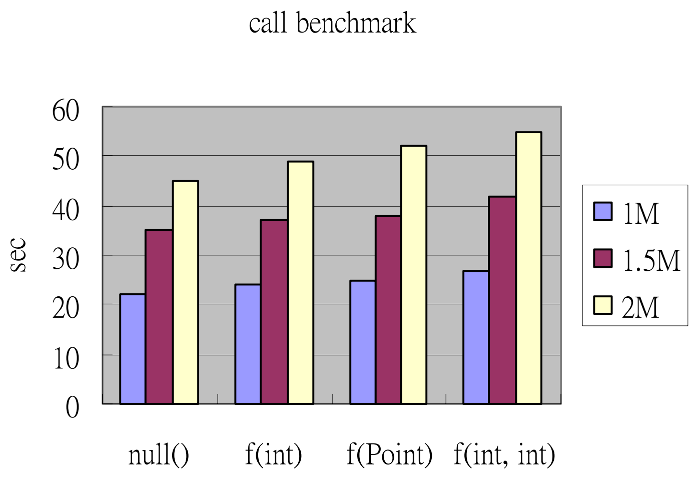
All benchmark was done with a server equipped with AMD Duron 1G cpu, 256MB ram running Redhat linux 7.2, a client machine of a IBM Thinkpad with Pentium III 600 cpu with 320MB ram running

Redhat linux 9 over 100Mbps Ethernet LAN. Both machine has the libasync library from SFS 0.7.2.

These evaluations prove that our project achieves a low overhead and a fairly good performance.

Call benchmark

In this evaluation, we used four different functions: 1. a null function; 2. one-integer parameter function; 3. one-Point object parameter; 4. two-integer parameter function. Their function bodies contained nothing there. The Point object contained only two integer values in it. These four function were each executed 1 million, 1.5 million and 2 million times. The time was measured in seconds. All benchmark was done on the server. The result of this evaluation was shown below.



We also calculated the average time per call for each function. The time was in micro seconds. The result was shown below and it was as we expected. Although the size of Point object was equal to two integer, there was still an obvious difference between f(Point) and f(int, int). The reason was that f(int, int) needs more function call and has overhead to handle the second integer parameter.

In addition, we also tested a null function call in Java RMI and calculated the average time per call. It turned out that our null function call of network object in C++ outperformed the one in Java RMI by 30 times. This evaluation proves that our implementation has a low overhead on function call and also has a good performance.

Method	Execution time (μ s)
null()	22.6
f(int)	24.4
f(Point)	25.4
f(int, int)	27.5
JavaRMI null()	701

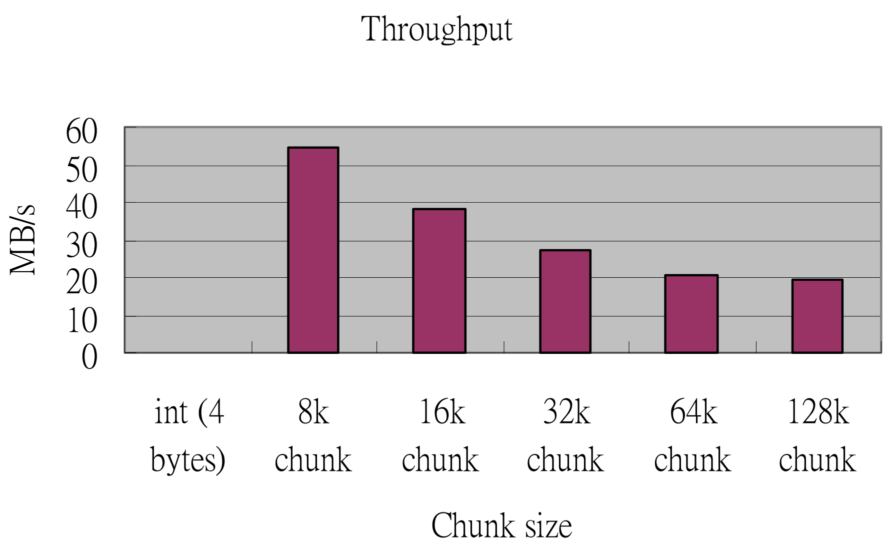
Throughput benchmark – wired version

The second evaluation we had done was the throughput benchmark over wire. We transmitted different numbers of packets from client to server over a 100Mbps Ethernet. Each packet had a payload chuck for 8k bytes. We transmitted 10k, 100k, 150k and 250k packets for each benchmark and recorded the time for transmission. The result was shown below. Obviously, the throughput was really closed to 12.5 MB/s, the theoretical maximum throughput of 100Mbps Ethernet. This evaluation provided information that our implementation as very efficient and easily saturated the 100Mbps Ethernet.

Packets	Time(s)	Throughput(MB/s)
10k	8	10.24
100k	73	11.22
150k	108	11.37
250k	181	11.31

Throughput benchmark – local version

The third evaluation we had done was throughput benchmark in local machine. We transmitted packets from different memory space on the server. In each test, we used different size of payload chunk of the packet. The sizes were 4, 8k, 16k, 32k, 64k and 128k bytes. Then we estimated the throughput value of each test. The result was shown below. From the result, we could observe that there was an overhead exists when the chunk size became larger. The chunk size became larger, the throughput approaches to one constant value. We could also notice that in 4 bytes (integer) argument, the overhead was so large that the throughput is below 1MB/s.



Limitation and possible solution

Our design has some limitations.

Since we compromise between simplicity, performance and full-feature pickling, our network object can not deal with non-simple object, which contains pointer or inherits from other object. This limitation should be easily resolved by introduced the approach two. By implementing the serialize interface, any object will be able to send over wire.

Second, all objects resides in the same server. Because we put the connection socket as the static member of Remote object, there is no way for individual proxy instance to connect to different machine. Moving the connection socket from static member into an instance stack should solve this problem.

Third, the server may have some resource leak if the client is dead or the connection is lost. We should be able to implement the garbage collector at the server to delete the orphan objects.

Moreover, the server is implemented in single-thread, asynchronous library, it could be blocked by a single function and rejects all other connections and tasks. We could spawn new thread for one connection and solve this problem. Using this multi-thread, asynchronous library should be a powerful solution.

And last, the pickling uses the memcpy() or bcopy() system call to deal with the memory duplication, there will be a potential problem if the client and server are running on different hardware. This problem arises when two machine use different byte-order. We could translate and always use big-endian format to pickle the object. Then the byte-order problem could be eliminated.

Conclusion

Our implementation simply achieves our expectation. The object is invoked remotely and working correctly. Besides, all primitive and simple object type could be used as arguments and transmitted over wire. The source code needs the least modification and hence proves the good transparency. At last, but not least, the performance is really great. We could easily saturate the 100Mbps Ethernet with CPU load around just 30% at server and 10% at client.

This implementation is fairly easy and could be done in a little more than one thousands line of codes. The architecture could be easily expanded with other feature thanks to the simple design.

With this project, we could easily customize any codes into the distributing environment and get rid of the error-prone low level network codes.

Appendix – Codes

The client program needs to include a proxy header file and add a line to connect to a specific server for a remote function invocation. Show in List 1.

List 1

```
#include <iostream>
#include “foo_proxy.h”

int main()
{
     Remote::connect(“localhost”, 3333);
     foo foo1;
     foo1.null();
}
```

In a proxy function, it simply encapsulates the arguments into a Variant object and aggregate all Variant objects into a ParameterList object. The proxy then call the Remote::invoke() to send the request over the wire. If there is a return value, it should Remote::invoke2() instead. Show in List 2.

List 2

```
foo::foo()
{
     ParameterList pl;
     create("foo()", pl);
}

foo::foo(int n)
{
     Variant arg1(n);
     ParameterList pl(arg1);
     create("foo(int n)", pl);
}

void foo::one(Point p)
{
     Variant arg1(p);
     ParameterList pl(arg1);
     invoke("one(Point p)", pl);
}

int foo::getInteger()
{
     ParameterList pl;
     return (int) invoke2("getInteger()", pl);
}
```

In the invoke function, Packet object is created and all the marshalling happens here. The return value is also received and passes back to the caller. Show in List3.

List 3.

```
const Variant
Remote::invoke2(const char* func_name, const ParameterList& pl)
{
    Packet packet(Packet::invoke, _id, typeid(*this).name());
    packet.func(func_name);
    packet.nParam(pl.count());
    packet.attachParam(pl);
    send(packet);

    recv(packet);
    assert(packet.act() == Packet::ok);

    return packet.pl()->variant(0);
}
```

The Packet will be parsed at the server and invoke the correct function by the object id and function signature. The pointer to the object will be cast back to the original object type and the arguments will be cast back to the original type and passed into the function. If there is a return value, it will be sent back in the form of ParameterList. Show in List4.

List 4

```
if(obj_name == typeid(foo).name() ) {
    foo *ptr = static_cast<foo*>(hash_entry.first);
    if( strcmp(packet.func(), "null()") == 0 ) {
        ptr->null();
    }
    else if( strcmp(packet.func(), "one(Point p)") == 0) {
        ptr->one(
           (Point) packet.pl()->variant(0)
           );
    }
    else if( strcmp(packet.func(), "getInteger()") == 0) {
        Variant ret = ptr->getInteger();
        ParameterList pl(ret);

        Packet ans(Packet::ok, 0, "");
        ans.func("");
        ans.nParam(pl.count());
        ans.attachParam(pl);
        send(ans);
    }
}
```

Reference

- Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber (Digital Systems Research Center), *Network Object*, 1993
- Vladimir Batov, *Implementing RMI for C++ Objects*, C/C++ User Journal 2003 March
- Sun Microsystem, *Java RMI v1.2*, 2003