# CORBA, DCOP and DBUS. A performance comparison.

## *Abstract*

For quite a while now I have been under the impression that the CORBA IPC/RPC mechanism used by the GNOME desktop environment was bloated and slow. People have commented that the DCOP and DBUS were speedier alternatives. I decided to put this supposition to the test by performing a simple set of experiments. The results were a little surprising.

## *Introduction.*

Remote procedure calls (RPC's) are function calls across process boundaries. In some cases these function calls will in fact be between computers, login sessions or between processes running within the one session. The common factor is that the calls are across the protection boundary maintained by the operating system between processes. This process boundary prevents applications from simply calling a function at a certain address within another process. The function call has to be "faked" in effect. The calling process (client) has to generate a message perhaps containing a function identifier and optionally some parameters and then send it to the other process (server). The server (which is presumed to be listening for messages) receives this message, parses it, passes the parameters to the specified function and returns a response message to the client. The exact mechanism by which message are sent varies greatly.

From a programmer's perspective, calling a properly encapsulated remote procedure should be no different to calling any other function. Inside the remote procedure call, messages will be exchanged across process boundaries, but other than perhaps taking a little longer to execute, the programmer should be unaware of any difference between a local and a remote procedure call. There are a range of technologies applicable to the "remoting" of function calls in Linux. Broadly speaking they all conform to the general form shown in Figure 1

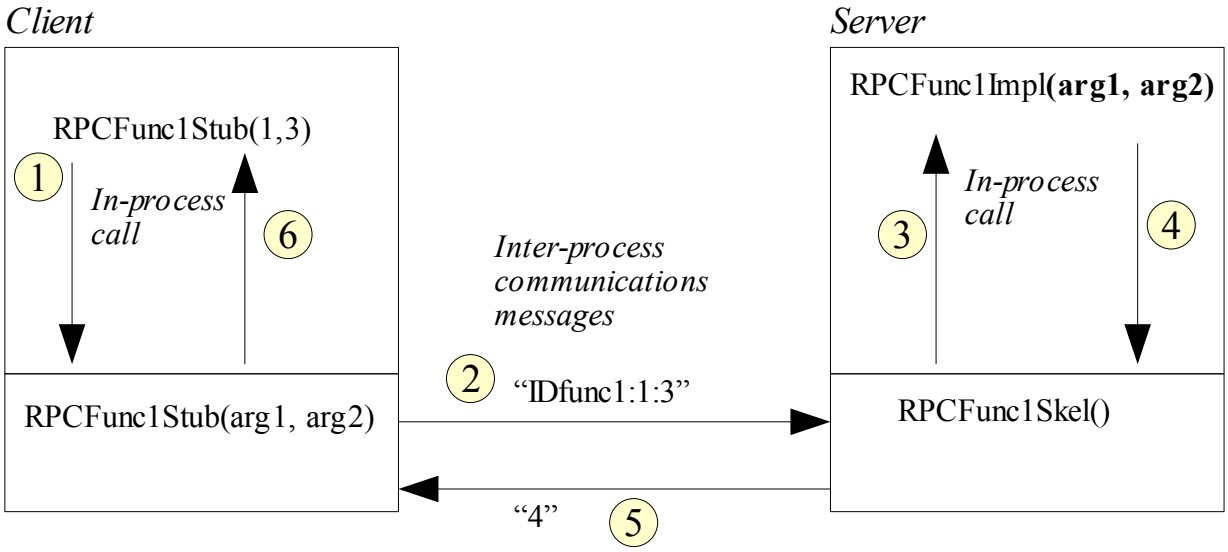


Figure 1

A client process calls on a stub function whose job is to generate an IPC message containing any parameters and some code to identify the remote function being sought. The stub also sends this message to the server.
The server includes a skeleton interface function whose job it is to extract parameters from inbound messages; route them to the correct function implementation (RPCFunc1Impl in Figure 1) and finally to format a reply message containing any results.
The stub hides the remote nature of the function from the client programmer. The skeleton hides the remote nature of the (service) function from the server programmer.

The different RPC technologies in Linux have different ways of formatting messages, locating servers, loading servers and so on but the overall form is as described here. It is of course possible to create custom message formats and to hand craft stubs and skeletons. There is however little to be gained from such an approach as there are a number of "middleware" tools and libraries available that automatically perform this function as well as providing additional support services.

## *CORBA, DCOP and DBUS*

Gnome (GNu Object Model Environment) uses CORBA for message passing. Gnome's implementation of CORBA primarily consists of an object request broker called ORBit and a component management library called Bonobo. CORBA is a well established inter object communications protocol standard maintained by the Object Management Group (OMG – see www.omg.org). There are many implementations of the various versions of this protocol standard for a range of different platforms. The basic goal however is to allow objects on different systems, developed in different languages to find and communicate with one another. Servers make objects available for use by clients. The object interfaces on a server are (usually) described using a special language used for describing object interfaces call Interface Definition Language (IDL).
DCOP/KDE uses its own interface definition language which alas is not the same as that commonly used for CORBA. It is similar to XML in structure. Finally DBUS uses XML to describe its client/server interfaces.

## *The test setup*

The IDL file that describes the interface for the CORBA benchmarking test is as follows:

```
interface benchmarking {
        void void_call(); // simple call with no parameters or results.
        void in_int_call(in long InValue); // one integer parameter sent to server
        void out_int_call(out long OutValue); // one integer result received from server
        void in_out_int_call(inout long InOutValue); // integer exchange in both directions
        void in_string_call(in string InValue); // one string parameter sent to server
        void out_string_call(out string OutValue); // one string result received from server
        void in_out_string_call(inout string InOutValue); // string exchange in both directions
};
```

This interface is used to test the speed with which strings and integers are passed back and forth between clients and servers.

The “kidl” file describing the DCOP interface is as follows:

```
<!DOCTYPE DCOP-IDL><DCOP-IDL>
<SOURCE> dcop_benchmarking_service.h </SOURCE>
<INCLUDE> qstringlist.h </INCLUDE>
<INCLUDE> qstring.h </INCLUDE>
<INCLUDE> dcopobject.h </INCLUDE>
<CLASS>
  <NAME> dcop_benchmarking_service </NAME>
   <SUPER> DCOPObject </SUPER>
   <FUNC>
     <TYPE> void </TYPE>
     <NAME> void_call </NAME>
   </FUNC>
  <FUNC>
      <TYPE> void </TYPE>
       <NAME> in_int_call </NAME>
     <ARG><TYPE> long int </TYPE><NAME> InValue </NAME></ARG>
   </FUNC>
  <FUNC>
     <TYPE>long int</TYPE>
     <NAME>out_int_call</NAME>
  </FUNC>
  <FUNC>
     <TYPE> long int </TYPE>
     <NAME> in_out_int_call </NAME>
     <ARG><TYPE> long int </TYPE><NAME> InValue </NAME></ARG>
   </FUNC>
  <FUNC>
     <TYPE> void </TYPE>
     <NAME> in_string_call </NAME>
     <ARG><TYPE> QString </TYPE><NAME> InValue </NAME></ARG>
  </FUNC>
  <FUNC>
     <TYPE> QString </TYPE>
     <NAME> out_string_call </NAME>
```

```
    </FUNC>
    <FUNC>
      <TYPE> QString </TYPE>
      <NAME> in_out_string_call </NAME>
      <ARG><TYPE> QString </TYPE><NAME> InValue </NAME></ARG>
    </FUNC>
</CLASS>
</DCOP-IDL>
```

The KDE/DCOP implementation differs from the GNOME/CORBA implementation in that Qstrings are exchanged instead of the simpler “char *” types. This was done mainly because QStrings are the “more native” string type for QT and KDE applications and in fact, they simplified the programming task.

The DBUS interface used for this benchmark test is described by the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<node name="/">
  <interface name="benchmarking.dbus.interface">
    <method name="void_call">
    </method>
    <method name="in_int_call">
          <arg type="i" name="arg1" direction="in" />
    </method>
    <method name="out_int_call">
          <arg type="i" name="result" direction="out" />
    </method>
    <method name="in_out_int_call">
          <arg type="i" name="arg1" direction="in" />
          <arg type="i" name="result" direction="out" />
    </method>
    <method name="in_string_call">
          <arg type="s" name="arg1" direction="in" />
    </method>
    <method name="out_string_call">
          <arg type="s" name="result" direction="out" />
    </method>
    <method name="in_out_string_call">
          <arg type="s" name="arg1" direction="in" />
          <arg type="s" name="result" direction="out" />
    </method>
  </interface>
</node>
```

## Results

In order to get a reasonable averaging effect, the time to call each of the functions 10000 was measured. The tests were carried out on the same system (Athlon XP 2800, 1GB RAM) running a copy of Fedora Core 5. Table 1 lists the results.

Table 1.

| *Function* | *CORBA execution time (microseconds)* | *DCOP execution time (microseconds)* | *DBUS execution time (microseconds)* |
|---|---|---|---|
| void_call | 626480 | 1769354 | 9783550 |
| in_int_call | 629227 | 1859219 | 10469005 |
| out_int_call | 660020 | 1824046 | 10399278 |
| in_out_int_call | 686850 | 1903499 | 11162058 |
| in_string_call | 650098 | 1902107 | 10510298 |
| out_string_call | 730118 | 1870306 | 10455126 |
| in_out_string_call | 682283 | 1952853 | 11239133 |

Analysis

A number of conclusions can be drawn from this.
The type of data being sent has only a marginal effect on the execution time.
DCOP is approximately 3 times ***slower*** than CORBA
DBUS is approximately 18 times ***slower*** than CORBA

These results were far from what I had expected. Why are DCOP and DBUS so slow? The answer may lie in the way that CORBA, DCOP and DBUS find particular functions on a server interface. DCOP and DBUS locate the desired function by passing its signature (a text-description of the function) to the remote server along with a block of data containing the parameters. CORBA on the other hand locates the desired function using pointers and an index.

Further analysis on the in_out_int_call was performed. This call is chosen because of its relative simplicity and because it exchanges data in both directions.

Note this further analysis was carried on a PC with the following specs:
Sempron 2600, 1GB RAM, Slackware 10.2 with Dropline Gnome.

## DCOP analysis

The DCOP stub on the client-side is as follows:

```
long int dcop_benchmarking_service_stub::in_out_int_call( long int arg0 )
{
    long int result = 0;
    if ( !dcopClient()  ) {
        setStatus( CallFailed );
```

```
        return result;
    }
    QByteArray data, replyData;
    QCString replyType;
    QDataStream arg( data, IO_WriteOnly );
    arg << arg0;
    if ( dcopClient()->call( app(), obj(), "in_out_int_call(long int)", data,
      replyType, replyData ) ) {
        if ( replyType == "long int" ) {
            QDataStream _reply_stream( replyData, IO_ReadOnly );
            _reply_stream >> result;
            setStatus( CallSucceeded );
        } else {
            callFailed();
        }
    } else {
        callFailed();
    }
    return result;
}
```

The following timings were observed:
Time to perform 10000 calls without modification : 2095730 microseconds
Time to perform 10000 calls with "if" block removed : 16651 microseconds
Time to perform 10000 calls with dcopClient()->call and no result processing : 2121165

From the above we can conclude that "dcopClient()->call" is the bottleneck. Delving into this a little it seems that each time this function is called, the service target function is "looked up" i.e. there is no caching of function id's/references.

## *DBUS Analysis.*

The call to in_out_int_call is as follows:

```
dbus_g_proxy_call (proxy, "in_out_int_call",&error,G_TYPE_INT, arg1,
G_TYPE_INVALID,G_TYPE_INT,&result, G_TYPE_INVALID);
```

This represents a direct call to dbus i.e. There are no stubs involved. Again, as is the case with DCOP there is a function lookup with no caching for subsequent calls.

## *CORBA analysis*

The CORBA stub for in_out_int_call looks like this:

```
void benchmarking_in_out_int_call(benchmarking _obj,
                                  CORBA_long* InOutValue,
CORBA_Environment *ev){

      gpointer _args[1];
      _args[0] = InOutValue;
      ORBit_c_stub_invoke (_obj, // the corba object
                           &benchmarking__iinterface.methods,
                           // address of // relevant method table
```

```
                        3,    // which method in that table
                        NULL, // return value
                        _args, // input arguments
                        NULL,  // context
                        ev,  // CORBA environment
                        benchmarking__classid, // unique class id
            G_STRUCT_OFFSET (POA_benchmarking__epv, in_out_int_call),
            (ORBitSmallSkeleton) _ORBIT_skel_small_benchmarking_in_out_int_call);

}
```

What may not be obvious from the above is that the target function is identified using pointers and an index. There is no potentially costly lookup.

Is there an underlying transport issue? This is not apparently so because the three systems appear to be based on unix stream sockets.

## *Conclusions.*

The main conclusion from the above is as follows:
For repeated calls to a particular RPC function, the C bindings to the Orbit2 orb outperform calls using the C++ bindings to DCOP and DBUS.

List of acronyms

| | |
|---|---|
| CORBA | Common Object Request Broker Architecture |
| DBUS | Desktop Bus |
| DCOP | Desktop Communications Protocol |
| IDL | Interface Definition Language |
| IPC | Inter process communication |
| GNOME | GNu Object Model Environment |
| OMG | Object Management Group |
| RPC | Remote Procedure Call |
| XML | Extensible Markup Language |