Performance Evaluation of Distributed Object Platforms for Telecommunications Service Engineering Activities

DIONISIS ADAMOPOULOS, GEORGE PAVLOU
Centre for Communication Systems Research (CCSR)
University of Surrey, Guildford, GU2 7XH
{D.Adamopoulos, G.Panlou}@eim.surrey.ac.uk

CONSTANTINE A. PAPANDREOU
Hellenic Telecom. Organisation (OTE)
17 Kalliga Street, GR-114 73, Athens, Greece
kospap@org.ote.gr

Abstract: - In response to major trends in the telecommunications market today and under influence of the emerging distributed computing technology the telecommunications industry is embracing distributed object platforms as a means enabling the successful participation in the open global services market of the foreseen era with new and advanced service offerings under increasing competition in a multi-vendor environment. In this realm, this paper presents an attempt to evaluate the performance of DCOM and CORBA under conditions which are common in telecommunications services engineered as distributed object applications. Finally, after the examination of important issues regarding the DCOM remoting architecture, some conclusions are drawn.

Key-words: - Distributed object platforms, DCOM, CORBA, new telecommunications services

1 Introduction

The telecommunications industry is currently facing a number of challenges imposed by changes in the telecommunications market. Deregulation, liberalisation, and competition imply requirements for higher utilisation of the network infrastructure, shorter time to market for new telecommunications services, much higher degree of customisation of these services, cost reduction of service development, open network provision, global connectivity, and global information access. Furthermore, both telecommunications networks and services are ever growing in sophistication and complexity with a tendency to become large-scale, highly decentralised and heterogeneous systems involving numerous users and resources. All these changes require more complex software systems and thus make evident the necessity to accelerate the integration of information technology and telecommunications.

Under these conditions the telecommunications industry is gradually adopting a new approach for the development, construction, and management of software for telecommunications services. This approach is characterised by the increased use of object-oriented Distributed Processing Environments (DPEs) as the infrastructure for new telecommunications services (telematic services) because they promise the benefits of more flexible service design and deployment, increased software reuse, and increased interconnection capabilities with external resources [1].

Currently, the two most important available distributed object platforms are Microsoft's (Distributed) Component Object Model (COM/DCOM) [2] and the Common Object Request Broker Architecture (CORBA), which is supported by the Object Management Group (OMG) [6]. Both of them assist service developers to cope efficiently with the complexity inherent in the process of realising telecommunications services as open distributed software applications comprised by heterogeneous service components which may be scattered across multiple organisations and distant locations [7].

In this paper, recognising the importance of distributed object platforms in telecommunications service engineering, the performance of DCOM and CORBA is examined with the intention to inform service designers and developers about the performance expectations that they should have when using these platforms, and thus assist them in a possible selection process between them.

2 Comparing DCOM and CORBA

Taking into account the basic characteristics of DCOM and CORBA, it is evident that they have similar architectures as both adopt a client / server based programming style and agree on the most fundamental aspects of their object models.

Service Engineering Related Properties	COM / DCOM	CORBA
Scalability	MTS, Active Dir. Service Interface (NT 5.0)	Naming service, Trader service
Reliability	MTS, MCS, MSMQ	Transaction service
Security	Built-in: NT LAN Manager, MTS, MS Crypto API, Authenticode SDK	Platform dependent: 3 security levels (0, 1, 2)
Manageability	MMC	Vendor specific tools, Transaction service
Support for Web-based Telematic Services	ActiveX, MS Active Server Page Technology	JavaScript/Java
Support for Internet/Extranets	Two-factor authentication, Remote Data Service (RDS)	Two-factor authentication, Secure Socket Layer (SSL)
Support for Intranets	Desktop tools, ActiveX, Active Data Objects, MSMQ	Desktop tools (via a bridge), Event & Persistence service

Table 1: Comparing DCOM and CORBA: Service engineering related properties.

In order to derive more general conclusions, compare the suitability and applicability of DCOM and CORBA in the telecommunications field, and gain an insight on the capabilities of these platforms pertaining their use in practical situations, a set of (telecommunications) service engineering related properties are identified, and their support by DCOM and CORBA is summarised in Table 1.

This table reveals that DCOM and CORBA differ in many respects and neither technology provides a complete solution for service engineering activities. However, both provide a solid infrastructure and there are specific scenarios in which each excels over the other.

3 Examination of Performance

The performance of distributed software, usually expressed in terms of execution time for a variety of operations / actions, is a critical factor for the development and provision of successful (user accepted and efficient) telematic services, especially when real-time functionality and / or multimedia characteristics are required. Therefore, a comparison between DCOM and CORBA has to address performance matters in order to be complete and offer a full insight of the practical value of the two most prominent distributed object platforms.

Performance comparisons between DCOM and CORBA that are found in the literature use a simple example of distributed code (two objects residing in two different computers that communicate via a network) and measure the time needed for the successful completion of a single remote method invocation (when one of the objects calls a method of the other object and waits for the result) [5]. Although such an approach is valid, a performance comparison based not just on a single remote method invocation, but on a series of logically related remote method invocations (forming a usage scenario / pattern) can lead to more accurate, reliable and illustrative results, and can also be the stimulus for the deduction of a number of more general, albeit useful and of practical value, performance related conclusions.

For this reason, in full agreement with the latter remark, an experiment involving multiple remote method invocations under both DCOM and CORBA was conducted. This experiment and the results obtained from it will be examined in the following paragraphs in an attempt to evaluate the performance of DCOM and CORBA under conditions which are common in telecommunications services engineered as distributed object applications.

Before proceeding, it has to be stressed that the performance of object-oriented DPEs, like DCOM and CORBA, that provide high-level network programming interfaces is comparable (under certain circumstances) with the performance experienced when using low-level, procedure-oriented, non-typesafe programming interfaces, such as BSD sockets [3]. Furthermore, the performance of both DCOM and CORBA keeps improving with the application of compiler optimisation techniques and the utilisation of light-weight communication protocols. Nevertheless, service developers seem to be willing to accept a certain performance penalty given all the benefits (and especially extensibility, maintainability, and reusability) they are gaining from using distributed object platforms [7].

3.1 The Experiment

A simple distributed object application implemented under both DCOM and CORBA constitutes the basis of the experiment that was conducted. More specifically, a server object returns (after an appropriate request) fixed length strings (each 80 characters long) to a client object in two different ways: one string after the other as a result of separate consecutive method calls, or by gathering a number of strings and returning them all together as a result of a single method call. Equivalently, the client object, when interacting with the server object, can either make multiple method calls for small amounts of data (one string) or a single method call for a larger amount of data (several strings). The IDL description of the server object interface in DCOM, which is similar to that in CORBA, is:

```
interface ITestServer : IUnknown
{ HRESULT GetSingleString([in] LONG index, [out] BSTR* item);
  HRESULT GetMultipleString([in] LONG index,
    [in] LONG count, [out] LONG* got, [out] BSTR** item);  };
```

The first method (`GetSingleString()`) of this interface returns a single string, based on the ID of that string that is included in `index`, as all strings are kept in an array until the data is requested. The second method (`GetMultipleString()`) returns a number of strings (`count`) starting at `index`. It has to be noted that in order to obtain comparable results, care was taken to have both the DCOM and CORBA versions of the testing code execute on the same operating system platform (MS Windows NT 4.0) and on exactly the same hardware and network infrastructure. To achieve this uniformity, the CORBA implementation of the client and server objects used Iona's Orbix ORB under MS Windows NT 4.0, and all the testing activity took place using two 350 MHz Pentium II computers with 64 MB of memory interconnected by a 10 Mbit/s Ethernet LAN. In the following paragraphs, important parts of the code of the client and server objects when implemented using DCOM will be presented and discussed. The CORBA implementation of this code retains the basic functionality and the only differences are those imposed by the special nature and characteristics of CORBA.

The server object has an embedded object of class `Elements`. When this object is created, it loads the strings in the specified file (`elements.dat`) and it makes each string 80 characters long by adding the appropriate number of dashes at the end of the string. Then, it keeps these strings in an array until the data is requested through the member function `GetItem()`. This function allocates system memory for the string at that index and returns it as a BSTR.

The `Elements` class is declared in `Elements.h`:

```
// Elements class
class Elements {
private:
   LPCTSTR* m_elements;
   LONG     m_size;
public:
   Elements(){};
   ~Elements();
   void Initialise(LPTSTR filename);
   BSTR GetItem(LONG ID);  };
```

It is implemented in `Elements.cpp`:

```
// Elements class
#include “stdafx.h”
#inlude <winnls.h>
#include “elements.h”
Elements::~Elements()
{  LONG index;
   for (index = 0; index < m_size; index++)
    HeapFree(GetProcessHeap(), 0, (LPVOID)m_elements[index]);
   HeapFree(GetProcessHeap(), 0, m_elements);  }
void Elements::Initialise(LPTSTR filename)
{  m_elements = NULL;
   m_size = 0;
   HANDLE hfile =
   CreateFile(filename, GENERIC_READ, 0, NULL,
              OPEN_EXISTING, 0, NULL);
   if (INVALID_HANDLE_VALUE == hfile) return;
   HANDLE hmapp = CreateFileMapping(hfile, NULL,
                  PAGE_READONLY, 0, 0, NULL);
   if (NULL == hmapp)
   {  CloseHandle(hfile); return;  }
   if (NULL == pstr)
   {  CloseHandle(hmapp); CloseHandle(hfile); return;  }
   m_size = 100;
   m_elements = (LPCTSTR*)HeapAlloc(GetProcessHeap(), 0,
                                    m_size * sizeof(LPCTSTR));
   if (NULL == m_elements)
   {  UnmapViewOfFile(pstr); CloseHandle(hmapp);
      CloseHandle(hfile); return;  }
   LPTSTR pos = pstr;
   LPTSTR next = pstr;
   DWORD size = GetFileSize(hfile, NULL);
   LONG count = 0;
   TCHAR padding[] =_T(“---------------------------------------
-------------------------------------------“);
   while (next < pstr+size)
   {  // Find end of string
      while (*next != _T(‘\r’) && *next != _T(‘\n’)
      && next < pstr + size) next++;
      // End of string, copy data
      LPTSTR temp = (LPTSTR)HeapAlloc(GetProcessHeap(),
                                      0, sizeof(TCHAR) * 160);
      lstrcpyn(temp, pos, next - pos + 1);
      lstrcat(temp, padding);
      temp[80] = _T(‘\0’);
      m_elements[count] = (LPCTSTR)temp;
      // Move to the next item
      count++; next++;
      if (_T(‘\n’) == *next) next++;
      pos = next;  }
   m_size = count;
   UnmapViewOfFile(pstr);
   CloseHandle(hmapp);
   CloseHandle(hfile);  }
BSTR Elements::GetItem(LONG ID)
{  if (ID >= m_size) return NULL;
   LPWSTR pstr;
#ifndef UNICODE
   pstr = (LPWSTR)HeapAlloc(GetProcessHeap(), 0,
                            sizeof(WCHAR) * 160);
   MultiByteToWideChar(CP_ACP, 0, m_elements[ID],
                       -1, pstr, 160);
#else
   pstr = m_elements[ID - 1];
#endif
   BSTR bstr = ::SysAllocString(pstr);
#ifndef UNICODE
   HeapFree(GetProcessHeap(), 0, pstr);
#endif
   return bstr;  }
```

In the case of `GetSingleString()`, a BSTR is generated from the string and passed back to the client:

```
HRESULT STDMETHODCALLTYPE CTestServer::GetSingleString(LONG
index, BSTR *item)
{  *item = m_list.GetItem(index);
   IF (NULL == *item)
   {  return S_FALSE;  }
   else
   {  return S_OK;  }  }
```

In the case of `GetMultipleString()`, an array is created, the BSTR for each item is placed in the array, and then the array and the number of items obtained are returned:

```
HRESULT STDMETHODCALLTYPE
CTestServer::GetMultipleString(LONG index,
                         LONG count, LONG* got, BSTR** item)
{  HRESULT hr = S_OK;
   // Temporary buffer
   BSTR* buf = new BSTR[count];
   *got = count;
   LONG current = index;
   while (current<(index + count))
   {  buf[current - index] = m_list.GetItem(current);
      if (NULL == buf[current - index])
      {  *got = current - index; hr = S_FALSE; break;  }
      current++;  }
   *item = (BSTR*)CoTaskMemAlloc(*got * sizeof(BSTR));
   for (current = 0; current < *got; current++)
      (*item)[current] = buf[current];
   delete [] buf;
   return hr; }
```

The client object determines whether the single or multiple case is used, specifies the start value and the number of items to get, and calculates the average time required to make the call:

```
void CTestClientDlg::OnGet()
{  CListBox* pList = (CListBox*)GetDlgItem(IDC_RESULTS);
   Assert(pList); pList->ResetContent();
   CWnd* pWnd = GetDlgItem(IDC_ELAPSE);
   Assert(pWnd); UpdateData();
   if (NULL == m_pTestServer)
   {  Message(_T(“The server interface pointer is NULL”));
      return;  }
   if (m_nRadio == 0)
   {  // Send multiple single requests
      BSTR* array = new BSTR[m_lCount];
      LONG count; LONG loop; LONG total = 0;
      for (loop = 0; loop < m_lRepeat; loop++)
      {  DWORD starttime = GetTickCount();
         for (count = 0; count < m_lCount; count++)
         {  HRESULT hr;
            hr = m_pTestServer->GetSingleString(count +
                                  m_lStart, &array[count]);
            if (FAILED(hr) || S_FALSE == hr)
            {  Cstring str;
               UpdateData();
               if (S_FALSE == hr)
                  str.Format("Cannot get item %ld",
                             count + mlStart);
               else
                  str.Format("Cannot connect: 0x%08x", hr);
               Message(str);
               break;  }  }
         DWORD endtime = GetTickCount();
         total += endtime - starttime;
         for (count = 0; count < m_lCount; count++)
         {  if (array[count])
            {  Cstring str(array[count]);
               ::SysFreeString(array[count]);
               if (loop == 0)
                  pList->AddString(str);  }  }  }
      Cstring str;
      str.Format("Duration %.21f milliseconds",
                 total/double(m_lRepeat));
      pWnd->SetWindowText(str);
      delete [] array;  }
   else
   {  // Send a single multiple request
      BSTR* array; HRESULT hr;
      LONG got; DWORD total = 0; LONG loop;
      for (loop = 0; loop < m_lRepeat; lopp++)
      {  DWORD starttime = GetTickCount();
         hr = m_pTestServer->GetMultipleString(m_lStart,
                             m_lCount, &got, &array);
         DWORD endtime = GetTickCount();
         total += endtime-starttime;
         if (FAILED(hr))
         {  UpdateData(); Cstring str;
            str.Format("Cannot connect: 0x%08x", hr);
            Message(str); break;  }
         LONG index;
         for (index = 0; index < got; index++)
         {  Cstring str(array[index]);
            ::SysFreeString(array[index]);
            if (loop == 0) pList->AddString(str);  }
         CoTaskMemFree(array);  }
      Cstring str;
      str.Format("Obtained %ld items", got);
      Message(str);
      str.Format("Duration %.21f milliseconds",
                 total/double(m_lRepeat));
      pWnd->SetWindowText(str);  }  }
```

This code examines whether this is a single or a multiple test. If it is a single test, the `GetSingleString()` method is called. Otherwise the `GetMultipleString()` method is called. `GetSingleString()` simply sends a single value and thus it is called for each of the required values (the number of strings and the first index are specified by the client). In order to increase the accuracy of the tests, the tests are repeated the number of times specified by the client. For the multiple case, the `GetMultipleString()` method is called on the server object just once. In both cases, the calls to the server object are timed and the average of the time to get the requested data is calculated.

3.2 The Results

During the experiment, two types of measurements were carried out, using both the DCOM and CORBA versions of the testing code. Initially, the client and the server objects were placed on the same machine, and the time (in ms) needed to transfer a number of strings from the server object to the client object, as a result of calling (on the server object), either `GetSingleString()` many times or `GetMultipleString()` once, was calculated for several different numbers of strings (Fig. 1). In this way, the performance of (the usually neglected) local method calls is examined under DCOM (in fact COM) and CORBA, recognising the fact that local object interactions are common even in large scale telematic services and thus they shouldn't be ignored or under-estimated. As can be seen in Fig. 1, local method calls are fast in both DCOM and CORBA, with DCOM being slightly faster than CORBA. Additionally, for both DCOM and CORBA, the single method call (`GetMultipleString()`) is about 10 times faster than making multiple method calls (`GetSingleString()`) for a specific number of strings.

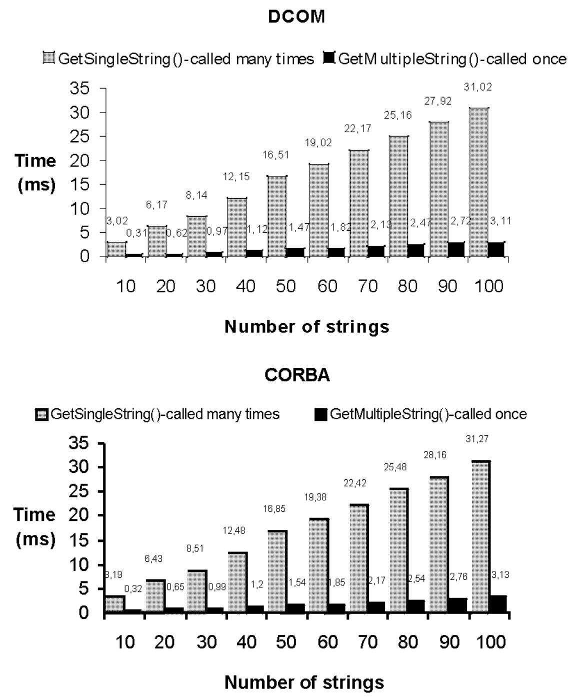
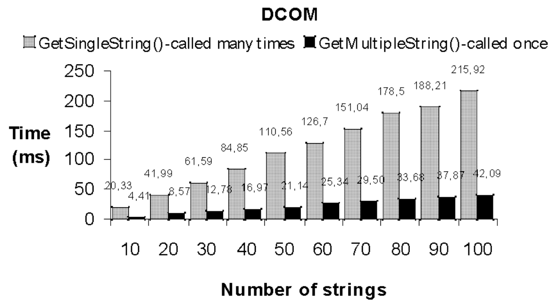


Fig. 1: Examination of local method calls in DCOM and CORBA.

The second type of measurements focused on remote method calls, which are the ones that affect mostly the performance of a telematic service. In this case, the time (in ms) needed to transfer a number of strings from the server object to the client object was calculated as in the first type of measurements, with the exception that the client and server objects were placed on two different machines connected via a network (Fig. 2). As would be expected, remote method calls take much longer than local method calls in both DCOM and CORBA, although the measurements have been taken on a very quiet network of just two machines running only the test software. From Fig. 2 is evident that CORBA is slightly faster than DCOM regarding remote object interactions, and that in both DCOM and CORBA, the multiple calls of GetSingleString() take about 5 times longer than the GetMultipleString() call for a specific number of strings.



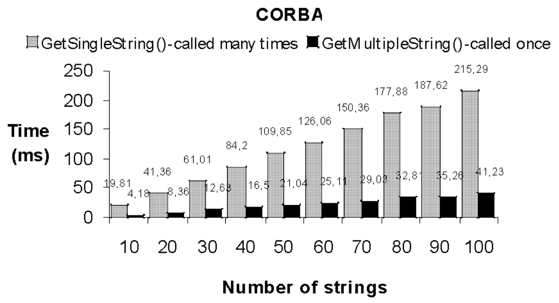


Fig. 2: Examination of remote method calls in DCOM and CORBA.

Taking into account all the measurements and evaluating the whole experiment it is apparent that in both DCOM and CORBA remote method calls are slower than local method calls, and many single method calls are five to ten times slower than a single multiple method call. Thus, in both these platforms, performance can be improved by placing objects (whenever possible) on the same machine that they will be used from, and by designing the interface of remote objects so that the corresponding (remote) method calls, which are required to perform a specific task, are kept to a minimum. Finally, Fig. 1 and 2 clearly illustrate that DCOM and CORBA have a comparable performance under the MS Windows operating system platform. Therefore, for this operating environment, a choice between DCOM and CORBA should not be based exclusively on performance considerations, but it should also take into account other more general and abstract / qualitative issues (see Section 2 and [1]).

This conclusion is reinforced even more by the fact that CORBA performance depends significantly on the implementation of the ORB by a specific vendor, and thus differs between different products. A similar situation is also true for DCOM, as DCOM's performance can be improved in certain circumstances by extending its remoting architecture which has built-in extensibility [4]. The way that this can be achieved is examined separately in the following section, because of the significant potential benefits it can offer to DCOM-based telecommunications services.

4 DCOM Remoting Architecture

Distributed object systems, such as DCOM and CORBA, provide the necessary infrastructure for supporting remote object activation and remote method invocation in a client-transparent way. The term remoting architecture refers to the entire infrastructure that connects clients to server objects [2].

A distributed object system does not necessarily have to specify how the entire remoting architecture should be structured. It can treat it as a black box as far as user applications are concerned. This approach has the advantage of allowing vendors to use their best performance optimisation techniques. However, a disadvantage is that such architectures are usually difficult to extend [9].

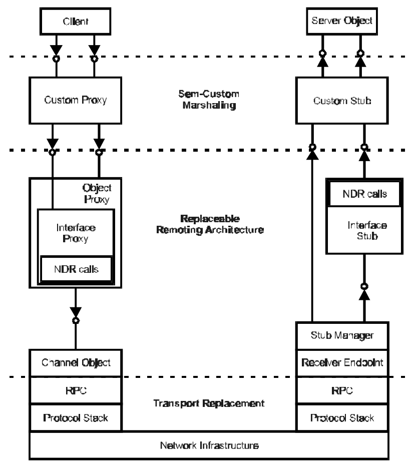


Fig. 3: The DCOM remoting architecture and extensibility options.

The DCOM remoting architecture can be seen in Fig. 3. Its main constituent parts are the following [2][4]:

- *Object proxies:* They act as the client-side representatives of server objects and connect directly to the client.
- *Interface proxies:* They perform client-side data marshaling and are aggregated into object proxies.
- *Client-side channel objects:* They use Remote Procedure Calls (RPCs) to forward the marshaled calls.
- *Server-side endpoints:* They receive RPC requests from clients.
- *Stub managers:* They are located in the server and they dispatch calls to appropriate interface stubs.
- *Interface stubs:* They perform server-side data marshaling and make actual calls on the appropriate server objects.
- *Standard marshaler:* It marshals interface pointers into object references on the server side and unmarshals the object references on the client side.

Usually, DCOM-based telecommunications services use standard marshaling. However, some of them may need to customise the client-server connection in order to express the correct semantics and improve performance. In these cases, the DCOM remoting architecture has to be extended.

The extensibility provided by DCOM can be divided into three categories; namely, *below*, *above*, and *within* [2][4][9]. The first category extends DCOM at the RPC layer and below, as shown in Fig. 3, in a way totally transparent to the standard remoting architecture.

To achieve the other two types of extensibility, DCOM supports a custom marshaling mechanism which allows a server object to bypass the standard remoting architecture and construct a custom one, optimised for a particular situation, without requiring source code modifications to the former. A server object declares that it wants to implement custom marshaling by supporting the IMarshal interface.

According to the second type of extensibility in DCOM, a handler layer can be insterted above the standard remoting architecture and below the user application (service components). This activity is often called semi-custom marshaling (or handler marshaling) because most of the tasks are eventually delegated to the standard remoting architecture, as shown in Fig. 3. As part of the marshaling / unmarshaling process, a custom proxy and a custom stub are inserted to allow additional processing of each method invocation.

The third type of extensibility in DCOM is the most general and the most promising one. According to it, as DCOM's remoting architecture is constructed at run time by instantiating and connecting various components, it should be possible for a custom architecture to reuse some of the binary components from the standard one and supply only the necessary custom objects. The construction of such a custom architecture is hard in current DCOM architecture, but it can be facilitated significantly by specialised architectures developed for this purpose [9].

CORBA does not specify a standard remoting architecture. Therefore, incorporating stronger system properties into CORBA-based telecommunications services and improving their semantics and performance is usually not done by exploiting the extensibility of the remoting architecture. Furthermore, while some CORBA-based systems allow the replacement of the marshaling code for a given interface (sometimes called smart proxies), DCOM is unique in that the remoting behaviour is polymorphically bound at runtime on an object-by-object basis, as two references of identical type may be using custom or standard marshaling independently. This allows object implementors to safely evolve their remoting implementation based on performance needs without rebuilding client applications.

5 Conclusions

The liberalisation of telecommunications markets has exposed service providers to a high level of competition. This competition is forcing them to reduce costs, improve customer service, and rapidly introduce new services. One key way in which these pressures can be addressed is through the increased exploitation of distributed object platforms.

In this paper, the performance of DCOM and CORBA, which currently are the two most important object-oriented DPEs, was examined focusing on their ability to support object interactions commonly used in new telecommunications services. The experiment that was conducted revealed that DCOM and CORBA have a comparable performance, although DCOM appears to be more flexible and with a significant potential for improved performance due to its extensible and customisable remoting architecture. However, there is no doubt that both DCOM and CORBA are important for the realisation of telecommunications services in today's heterogeneous information networking environment. Therefore, their efficient interoperation via a standardised single two-way gateway specification (a bridge) between them is expected to rapidly gain importance [8].

References

[1] Adamopoulos, D.X., Pavlou, G., Papandreou, C.A., "Distributed Object Platforms in Telecommunications: A Comparison Between DCOM and CORBA", *BT Engineering*, Vol. 18, 1999, pp. 43-49.

[2] Brown, N., Kindel, C., "*Distributed Component Object Model Protocol - DCOM*", Microsoft Corporation, January 1998.

[3] Fatoohi, R.A., "Performance Evaluation of Communication Software Systems for Distributed Computing", *Distributed Systems Engineering*, No. 4, 1997, pp. 169-175.

[4] Grimes, R., "*Professional DCOM Programming*", Wrox Press, 1997.

[5] Lewandowski, S.M., "Frameworks for Component-Based Client/Server Computing", *ACM Computing Surveys*, Vol. 30, No. 1, 1998, pp. 1-27.

[6] Object Management Group, "*The Common Object Request Broker: Architecture and Specification*", Revision 2.0, July 1995.

[7] Saleh, K., Probert, R., Khanafer, H., "The Distributed Object Computing Paradigm: Concepts and Applications", *The Journal of Systems and Software*, No. 47, 1999, pp. 125-131.

[8] Smith, G., Gough, J., Szyperski, C., "A Case for Meta-Interworking: Projecting CORBA Meta-Data into COM", *Proceedings of TOOLS '98*, Nov. 1998.

[9] Wang, Y.-M., Lee, W.-J., "COMERA - COM Extensible Remoting Architecture", Proceedings of COOTS '98, 1998, pp. 79-88.