Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks

Rajesh S. Madukkarumukumana, Hemal V. Shah
Intel Corporation
Calton Pu
Oregon Graduate Institute of Science and Technology

Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks

Rajesh S. Madukkarumukumana
Server Architecture Lab
Intel Corporation
5200 N.E. Elam Young Pkwy
Hillsboro, OR 97124
rajesh@co.intel.com

Calton Pu
Department of Computer Science & Engineering
Oregon Graduate Institute (OGI) of Science and Technology
Portland, OR 97291
calton@cse.ogi.edu

Hemal V. Shah
Server Architecture Lab
Intel Corporation
5200 N.E. Elam Young Pkwy
Hillsboro, OR 97124
hvshah@co.intel.com

Abstract

In a distributed object system such as Distributed Component Object Model (DCOM) [5, 7], legacy transport protocols used for communication limit the performance over high-speed networks. By making use of a low-latency, high-bandwidth, and low overhead user-level networking architecture such as Virtual Interface (VI) Architecture [8, 18], this performance bottleneck can be significantly reduced. Since user-level networking architectures provide low-level primitives, the challenge lies in integrating them into high-level applications. This requires a systematic approach. In this paper, a methodology to utilize VI Architecture to improve the performance of DCOM using custom object marshaling is developed. Initial experimental results demonstrate that the latencies of small messages in distributed object computing can be significantly reduced by this methodology.

Keywords: Virtual Interface (VI) Architecture, User-level Networking Architecture, Distributed Component Object Model (DCOM), Distributed Object Computing, Custom Object Marshaling.

1. Introduction

Component based software offers modularity, reduces applications' integration and maintenance costs, and improves deployment flexibility. Distributed object frameworks like Distributed Component Object Model (DCOM) [7], Common Object Request Broker Architecture (CORBA) [16], and Java Remote Method Invocation (RMI) [13] facilitate building distributed applications from simple components. Distributed object frameworks use remote procedure call (RPC) mechanism to perform remote object activations and remote method invocations. The overheads associated with underlying legacy transport protocols (e. g. UDP, TCP) used in RPC mechanisms introduce considerable latency over high-speed networks such as System Area Networks (SANs).

User-level networking architectures, such as the Virtual Interface (VI) Architecture [8, 18], U-Net [10], and SHRIMP Virtual Memory Mapped Communication (VMMC) [2] that are designed to achieve low-latency and high-bandwidth in a SAN environment, offer an attractive solution for reducing communication software overheads. Building high-level applications, using low-level primitives offered by user-level networking architectures, is complex. This paper focuses on the challenge in integrating user-level networking architectures into distributed object frameworks. In this research, DCOM is the target distributed object model and VI Architecture is used as the user-level networking architecture. This paper provides the following two contributions in harnessing user-level networking architectures for distributed object computing over high-speed networks:

- A specialization methodology to replace legacy RPC transports in DCOM with VI-based transport for SAN environments,
- Latency analysis of standard and VI-enabled DCOM remoting architecture.

Integration of VI into DCOM remoting architecture is achieved by custom object marshaling mechanism. This involves specialization of object implementation and generation of custom proxy/stub code along with marshaling routines. Initial experimental results provide evidence of the performance improvement.

The organization of the rest of the paper is as follows. Brief overviews of VI Architecture and DCOM are provided in Section 2 and Section 3 respectively. In Section 4, a mechanism to integrate VI Architecture into DCOM to reduce remote method invocation latencies is discussed. Experimental results are provided in Section 5. Section 6 briefly summarizes some related work. Finally, future work is discussed and conclusion is drawn in Section 7.

2. Virtual Interface Architecture

VI Architecture is a user-level networking architecture designed to achieve low latency, high bandwidth communication between processes running on nodes connected by a high-speed network within a computing cluster. To a user process, the VI Architecture provides direct access to the network interface in a fully protected fashion. The VI Architecture avoids intermediate copies of the data and bypasses operating system to achieve low latency, high bandwidth data transfer. The VI Architecture Specification 1.0 [18] was jointly authored by Intel Corporation, Microsoft Corporation, and Compaq Computer Corporation.

The VI Architecture uses a VI construct to present an illusion to each process that it owns the interface to the network. A VI is owned and maintained by a single process. Each VI consists of two work queues: one send queue and one receive queue. On each work queue, Descriptors are used to describe work to be done by the network interface. A linked-list of variable length Descriptors forms each queue. Ordering and data consistency rules are only maintained within one VI but not between different VIs. VI Architecture also provides a completion queue construct that is used to link completion notifications from multiple work queues to a single queue.

Memory protection for all VI operations is provided by protection tag (a unique identifier) mechanism. Protection tags are associated with VIs and memory regions. The memory regions used by Descriptors and data buffers are registered prior to data transfer operations. Memory registration gives VI NIC a method to translate virtual addresses to physical addresses. The user receives an opaque memory handle as a result of memory registration. This allows user to refer to a memory region using a memory handle/virtual address pair without worrying about crossing page boundaries and keeping track of the mappings of virtual addresses to tags.

The VI Architecture defines two types of data transfer operations: 1) traditional send/receive operations, and 2) Remote-DMA (RDMA) read/write operations. A user process posts Descriptors on work queues and uses either polling or blocking mechanism to synchronize with the completed operations. The two Descriptor processing models supported by VI Architecture are the work queue model and the completion queue model. In the work queue model, the VI consumer polls or waits for completions on a particular work queue. The VI consumer polls or waits for completions on a set of work queues in the completion queue model. The processing of Descriptors posted on a VI is performed in FIFO order but there is no implicit relationship between the processing of Descriptors posted on different VIs.

For more details on VI Architecture, the interested reader is referred to [8, 18]. Figure 2 compares the one-way latency of UDP with one-way latency of software emulated VI (in host driver) over a 100 Mbps Ethernet. The latencies were measured using ping-pong tests and were averaged over 1000 runs. Figure 2 illustrates that even the latency of VI emulated in host driver is significantly less than the latency of UDP. With VI functionality implemented in NIC hardware, the latency can be significantly reduced further.

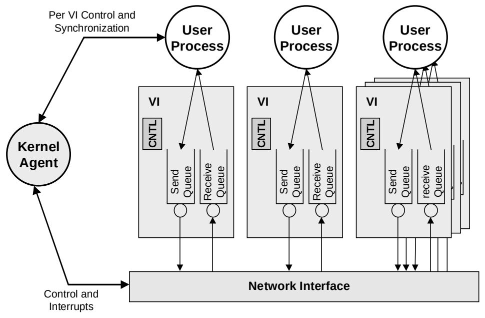


Figure 1: VI Queues

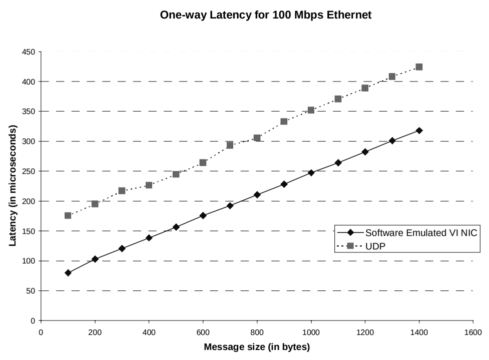


Figure 2: Emulated-VI vs. UDP Latency

3. Distributed Component Object Model (DCOM)

The Component Object Model (COM) [5] is an architecture and a supporting infrastructure for creating, using and evolving component software and building applications using these components. COM provides a binary standard to which components and its clients must adhere in order to ensure dynamic interoperability. Distributed Component Object Model (DCOM) [7] is an extension to COM for networked environments to support distributed computing. The overall DCOM architecture consists of the COM programming interface, the interface remoting infrastructure, and the wire protocol. COM allows clients to communicate with an object solely through the use of *vtable*-based interface instances. This provides a single programming model for accessing in-process, local and remote components. The interface remoting infrastructure in COM facilitates this location transparency. The DCOM wire protocol describes the content and the format of what is actually transmitted across the network when components reside on remote machines.

3.1 DCOM Architecture

The marshaling architecture in DCOM performs encoding and decoding of method call/return parameters into a standard data representation (marshaling and unmarshaling) that can be sent across the network. DCOM remoting architecture is abstracted as an Object RPC (ORPC) layer built on top of DCE RPC infrastructure. DCE RPC defines the standard data representation (NDR) for all relevant data types.

Interface pointers in COM are either returned from object activations or passed as parameters in method calls. COM has a special data type not present in DCE RPC to handle interface pointers in a uniform way. Marshaling and unmarshaling of COM interface pointers entails creation of a stub object in the server process and a proxy object in the client process respectively. Proxy and stub are capable of handling remote method invocations to the marshaled interface.

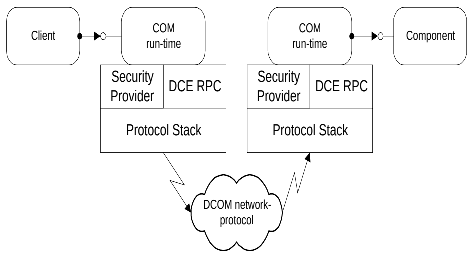


Figure 3: DCOM Architecture

The COM Specification defines various types of interface pointer marshaling, namely, standard marshaling, handler marshaling, and custom marshaling. Standard marshaling in COM provides the glue to the underlying RPC infrastructure and allows the component and the client to be completely ignorant of the marshaling and remoting architecture. Compiling the component's Interface Description Language (IDL) file with the MIDL compiler generates the proxy and stub code for standard marshaling. Handler marshaling extends the COM marshaling architecture by allowing the component to plug-in smart handlers that can intercept client's method calls and choose to satisfy them or forward them to the standard proxy. The design of an interface that focuses only on its function can lead to design decisions that conflict with efficient implementation across a network. In cases like these, COM allows object implementers to extend or even override standard marshaling of an interface pointer by the use of custom marshaling. Custom marshaling maintains complete client transparency. This architectural extensibility makes it possible to address network performance issues without disrupting the established design. For more details on COM and DCOM architectures, the interested reader is referred to [5, 7].

Custom marshaling allows the object to dynamically choose how its interface pointers are marshaled. Custom object marshaling is useful in many techniques including:

- replacing COM ORPC with other transports,
- marshaling static objects by value,
- adding fault-tolerance and high-availability properties to objects,
- performing replication transparently to the client and the component.

Wang et. al. briefly described some of these techniques in [19]. RPC infrastructure used in COM standard marshaling can work over a variety of legacy transport protocols like UDP, TCP, etc. Due to the inherent scalability offered by UDP, it is the default (and most widely used) DCOM protocol. Figure 4 shows the one-way latency (averaged over 1000 runs) of COM, RPC and UDP measured using ping-pong tests. DCOM and RPC measurements used bi-directional conformant arrays with the following method signature:

```
HRESULT MoveData (
      [in] ULONG ArraySize,
      [in, out, size_is(ArraySize)]  ULONG *pArray );
```

The measurements clearly show that for small messages (common case in distributed object computing frameworks), latency incurred in RPC and DCOM is dominated by UDP latency.

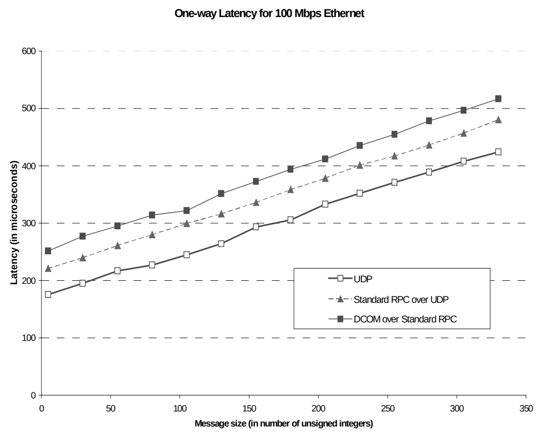


Figure 4: UDP, RPC, and DCOM Latency

4. DCOM Remote Method Invocation over VI Architecture Transport

COM marshaling architecture is extensible through its implementation of proxies and stubs as in-process COM servers. A COM object implementation advertises its ability to perform custom marshaling by exporting a standard COM interface called *IMarshal*. An object that does not export the *IMarshal* interface gets the standard proxy and stub by default. As part of marshaling an interface pointer, COM allows objects to perform any arbitrary action (like creating a custom stub) and to provide any block of data representing the custom object reference. The object can also specify the class identity (CLSID) of the custom proxy that can unmarshal the custom object reference on the client side.

Upon receiving the marshaled data, COM runtime instantiates the specified custom proxy in the client process. The custom proxy uses the marshaled object reference data to setup a connection to the stub and exposes the same *vtable* representation of the remoted interface to the client. Figure 5 illustrates a custom marshaling architecture that uses the high performance user-level VI transport for inter-process communications. The architectural details are described next.

4.1 Object Specialization using *IMarshal*

To enable COM remoting over VI transport, the object implementation needs to be specialized to expose the standard *IMarshal* interface. This is achieved by performing a source-to-source transformation of the object implementation. COM supports the notion of composing an object from binary composites using a component re-use technique called aggregation [5]. COM aggregation is useful in specialization as it allows composing objects dynamically. To minimize the source transformation needed to expose the *IMarshal* interface, the specialized object aggregates *IMarshal* from the inner custom stub. The specialization process is automated by a *"Custom Marshaling Wizard"* integrated into the Microsoft Developer Studio environment as a *"DevStudio Add-In"* component.

In the current prototype, COM automation interfaces, non-C++ object implementations, and VI Remote DMA (RDMA) operations are not supported. In order to support co-existence of standard and custom remote proxies and to preserve object identity, the available context information needs to be extended by using either ‘channel hooks’ [9] or custom class factories. Security features are not present in the current custom marshaler, but can be provided using standard Windows NT challenge/reply authentication.

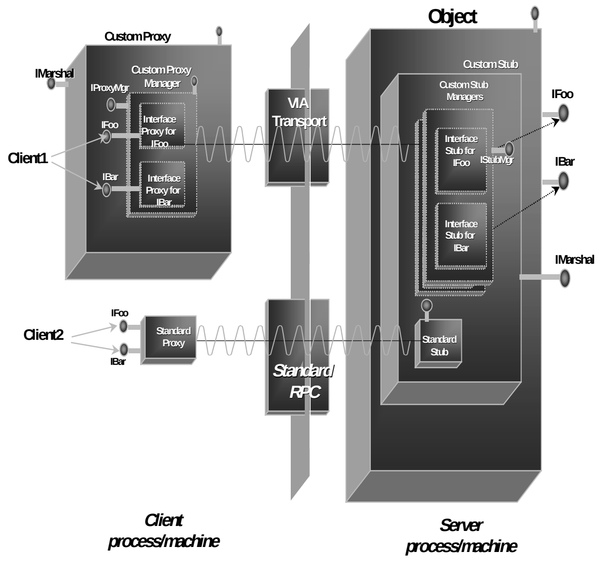


Figure 5: Custom Object Marshaling

4.2 Anatomy of Custom Stub/Proxy

A custom stub (proxy) is itself a COM object. A custom stub consists of a stub manager and interface stubs for each of the component's marshaled interfaces. Each custom stub manager represents an endpoint connection from a specific remote client process to the marshaled object. A custom stub manager manages endpoint creation and destruction, data transfers, and object lifetime. It also dispatches method requests to interface stubs. The custom stub uses the context and marshal flags passed as *IMarshal* method parameters to delegate unsupported contexts (e.g. table marshaled and local objects) to the standard marshaler. Each interface stub unmarshals method parameters from receive buffers, dispatches actual object methods, marshals return parameters into the reply buffers, and returns to the stub manager. The stub manager sends the reply buffer to the client. On the client side, each custom proxy is a peer to the corresponding custom stub and consists of proxy manager and interface proxies.

Marshaling method parameters into the standard data representation (NDR) provides heterogeneity and allows application programmers to use any user-defined data structures as method parameters. NDR marshaling (*Pickling*) used by the custom proxies and stubs avoids intermediate buffer copies by marshaling method parameters directly into registered transmission buffers. Procedural encoding is used to avoid buffer packing complexities, and incremental encoding is used to meet dynamic memory requirements. The NDR routines are generated from a transformed IDL file derived from the original application IDL file using the following rules:

1. Each method is split into a request method and a reply method. The request method contains all the method parameters passed from the client to the object. The reply method contains parameters returned from the object to the client including the HRESULT. Any parameter that is used in attributes of return parameters is also included in the reply method.
2. Since request and reply method parameters are marshaled at one end and unmarshaled at the other end, each parameter is declared bi-directional and a level of indirection is added to it. The added indirection is propagated in parameter attributes as well.
3. Marshaling of interface pointers in method parameters are handled separately as Windows™ NT encoding services do not currently support them.
4. Custom proxies and stubs support a set of special interfaces to allow marshaling of interface pointers that are references to custom proxies.

The following example shows an IDL transformation using some of the above rules.

```
Original Method's Signature:
    HRESULT MoveData(
        [in] ULONG ArraySize,
        [in, out, size_is(ArraySize)]
        ULONG *pArray );

Transformed Methods' Signatures:
    void MoveData_Request(
        [in, out]  ULONG *ArraySize,
        [in, out, unique, size_is(, *ArraySize)]
        ULONG **pArray );
    void MoveData_Reply(
        [in, out] HRESULT *ReturnCode,
        [in, out]    ULONG *ArraySize,
        [in, out, unique, size_is(, *ArraySize)]
        ULONG **pArray );

Generated NDR Routines:
    void MoveData_Request(
        handle_t IDL_handle,
        ULONG *ArraySize,
        ULONG ** pArray );
    void MoveData_Reply(
        handle_t IDL_handle,
        HRESULT *ReturnCode ,
        ULONG *ArraySize,
        ULONG **pArray );
```

By running the MIDL compiler over the transformed IDL file along with a supporting application configuration file (ACF), the NDR encoding routines are generated. The custom proxy (stub) dynamic link library (DLL) is created from the generated NDR routines, interface proxy (stub) templates, and the static proxy (stub) manager code. The whole process of custom proxy and stub generation can be automated by integrating it into the *"Custom Marshaling Wizard"*. Figure 10 provides a snapshot of *"Custom Marshaling Wizard"*.

5. Experimental Results

In order to demonstrate DCOM performance improvements achieved by integrating user-level VI transports, a set of experiments was carried out. In the experiments, a pair of server systems, with dual 200 MHZ Pentium ® Pro processors (with 256K L2 cache), Intel 82440FX PCI chipset, and 64 MB memory, was used as a pair of host nodes. Intel Pro100B Ethernet (100 Mbps) NIC with VI functionality emulated in software (host driver), Myricom's Myrinet [3] NIC (1.28 Gbps) with VI functionality emulated on NIC firmware, and GigaNet's cLAN™ GNN1000 interconnect (1.25 Gbps full duplex) [11] with VI functionality implemented on NIC hardware were used as VI NIC prototypes. The software environment used for all the experiments included Windows™ NT 4.0 with service pack 3 and Microsoft Visual C++ 5.0.

VI and UDP latency tests measure the time to copy the contents from an application's data buffer to another application's data buffer across an interconnect using a round-trip (ping-pong) test. DCOM and RPC latency measurements used bi-directional conformant arrays as method parameters with the method signature described in Section 4.2. VI architecture provides both polling and blocking models for synchronization. In the polling model, the user thread directly polls on the status of descriptors posted on VI work queues, thereby avoiding interrupt generation and processing overheads at the cost of increased CPU utilization. Reducing interrupts has a significant impact on the capacity of the system in addition to reducing the per-packet send/receive latencies. For GigaNet VI NICs, the experiments were carried out for both polling and blocking models. In the experiments involving Ethernet and Myrinet, only polling model was used.

In all the experiments, COM servers and clients used were free-threaded and COM security features were disabled. In case of custom stubs and proxies, method parameters and other information (including the NDR header) are marshaled and unmarshaled directly into and out of registered send/receive communication buffers to avoid intermediate data copies.

The VI Architecture is designed to enable applications to communicate over a SAN that provides high bandwidth, low latency communication with low error rates. At the NIC level, the VI Architecture provides three levels of reliability: Unreliable Delivery, Reliable Delivery, and Reliable Reception. Only VIs with the same reliablility level can be connected. In the experiments, the level of reliability used in each VI was Reliable Delivery. According to [18], this level of reliability guarantees that all data submitted to a reliable delivery VI will arrive at its destination exactly once, intact, and in the order submitted, in the absence of errors. For this level of service, transport errors are considered catastrophic and should be extremely rare. Due to this level of service used along with low error rates on high-speed networks, error recovery in form of application specified timeouts was incorporated in custom marshaled proxies and stubs.

Figures 6 and 7 compare one-way COM remote method invocation latencies (averaged over 1000 runs) between the standard and the specialized components across Ethernet and Myrinet interconnects respectively. Since the VI functionality was emulated either in host driver or in NIC firmware, the performance measurements are conservative.

From Figures 6 and 7, it is clear that VI-based communication in DCOM substantially reduces the performance bottleneck due to the use of legacy protocol stack. In the case of Ethernet, one-way latency was reduced by more than 150 microseconds (30% - 60 %) by using VI-based communication in DCOM. Due to unavailability of the standard Windows™ NT NDIS driver on Myrinet, UDP-based measurements were not obtained. Interestingly, Figure 7 indicates that on Myrinet the performance of DCOM over VI is better than that of COM over local RPC for small messages (≤ 2 KB). The VI-emulation in Ethernet driver is useful for proof-of-concept validation, but it achieves only limited performance. Even though the VI-emulation on Myrinet NIC performs better, the slow (33MHZ) on-board controller (MCP) limits the overall gain.

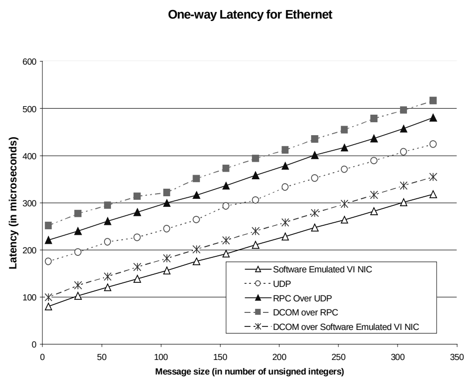


Figure 6: DCOM over VI on Ethernet (Polling Model)

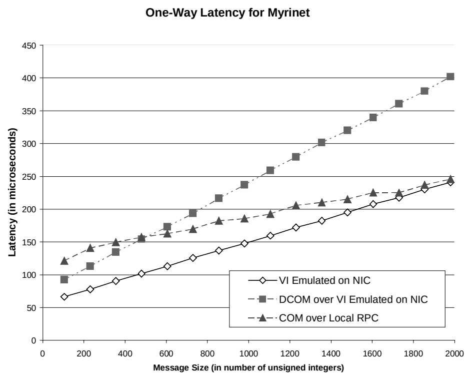


Figure 7: DCOM over VI on Myrinet (Polling Model)

Figures 8 and 9 show the results of similar experiments over the GigaNet cLAN™ GNN1000 native VI NICs using blocking and polling synchronization models respectively. The results obtained over GigaNet VI NICs confirm that availability of core VI functionality in special purpose hardware on network adapters can significantly improve communication performance. Figures 8 and 9 also demonstrate that the specialization methodology developed in this paper can deliver the raw performance offered by these high-speed interconnects to higher level COM applications.

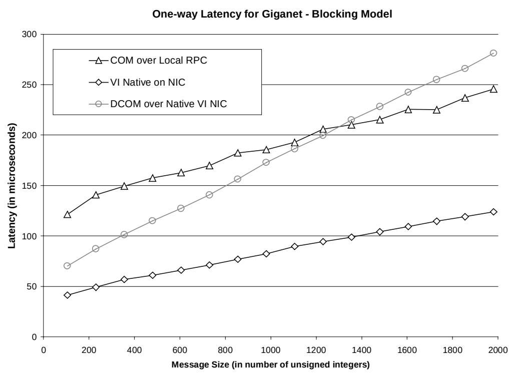


Figure 8: DCOM over VI on GigaNet (Blocking Model)

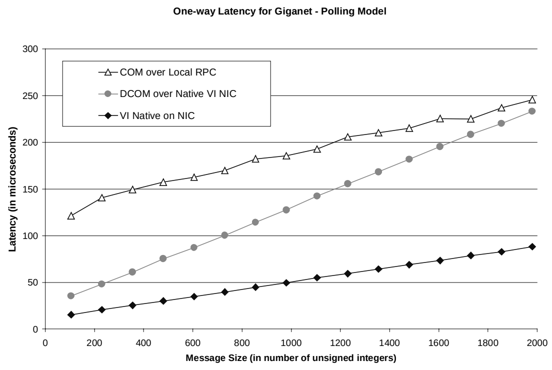


Figure 9: DCOM over VI on GigaNet (Polling Model)

The NDR processing continues to be a significant part of the remaining COM remote method invocation overhead (this was verified by using Intel VTune™ performance monitoring tool) and is a good candidate for further optimization using other specialization techniques like partial evaluation as proposed by Muller et. al. [15]. Figures 8 and 9 also indicate that the DCOM performance over GigaNet VI NICs is better than the performance of COM over local RPC for small messages (≤ 5 KB for blocking model and ≤ 8 KB for polling model).

For both GigaNet and Myrinet, DCOM over VI outperformed COM over local RPC for small messages. In case of COM over local RPC, communication between proxy and stub involves overheads of context switching and synchronization. On the other hand, since proxy and stub reside on different nodes in DCOM, the cost of context switching between proxy and stub is eliminated. The advantage of VI Architecture here is that the VI NIC performs the tasks of multiplexing, demultiplexing, and data transfer scheduling normally being performed by an OS kernel and device driver in legacy transports. Thus, OS overheads are significantly reduced in case of DCOM over VI. This suggests that for small messages, with high-speed interconnects and low overhead user-level networking architecture like VI Architecture, the cost of moving data between two processes residing on different nodes can be less than the cost of moving data between two local processes residing on the same node.

6. Related Work

Application level optimizations such as application level framing and integrated layer processing are used by Schmidt et. al. [12] to reduce CORBA latency. COMERA [19] proposes an extensible COM remoting architecture for transparent fault tolerance, migration, and replication properties. Quarterware kit [17] enables building middleware implementations that can be customized for performance and additional features. Muller et. al. [15] showed how specialization techniques like partial evaluation can be applied to improve RPC performance. All of the above approaches use application level optimizations but do not address utilizing user-level networking architectures to improve performance. Damianakis et. al. [6] pointed out that performance of higher level programming models such as stream sockets and remote procedure calls can be improved by using a user-level networking architecture. This paper adds on to their findings by improving object middleware performance using VI Architecture.

7. Future Work and Conclusion

Object specialization through custom object marshaling requires modifications to the object implementation, even though this can be fully automated. By adding a new protocol sequence for VI Architecture in addition to current suite of RPC protocols (ncadg_ip_udp, ncacn_ip_tcp, etc.) complete client and object transparency can be achieved. We are currently investigating this approach.

In this paper, custom marshaling based object specialization methodology was developed to integrate high-performance user-level networking architectures into distributed component object model (DCOM). The experimental results confirm that high-performance provided by VI Architecture can be delivered to high level COM applications using this specialization methodology. Standard high volume servers (SHVs), commodity high-speed interconnects, and standard based user-level networking architectures like VI Architecture can open new horizons to off-the-shelf distributed applications by providing high performance at low cost.

Acknowledgements

We would like to acknowledge David Fair, Roy Larsen, Wire Moore, Greg Regnier, and Mitch Shults of Intel Corporation for their many useful suggestions and reviews.

References

1. Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber, Network Objects, DEC SRC Research Report 115.

2. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, " A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer", *Proc. of the 21st Annual Symposium on Computer Architecture*, 1994, pp. 142-153.

3. N. J. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and Wen-King Su, "Myrinet: A Gigabit-per-Second Local-Area Network", *IEEE MICRO*, Vol. 15, No. 1, February 1995, pp. 29-36, http://www.myri.com/research/publications/Hot.ps

4. P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. Shih, C.-Y. Wang, and Y.-M. Wang, "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer", *C++ Report*, January 1998.

5. Component Object Model Specification, Microsoft Corporation, 1995. http://www.microsoft.com/com

6. Stefanos Damianakis, Angelos Bilas, Cezary Dubnicki, and Edward W. Felten, "Client Server Computing on SHRIMP", *IEEE MICRO*, January/February 1997, Vol. 17, No. 1.

7. DCOM Architecture, Microsoft Corporation, 1997.

8. Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, Ellen Deleganes, David Fair, Chris Dodd, and Justin Rattner, "The Virtual Interface Architecture: A Protected, Zero Copy, User-level Interface to Networks", *IEEE MICRO*, March/April 1998, Vol. 18, No. 2, pp. 66-76.

9. Guy Eddon and Henry Eddon, "Understanding the DCOM Wire Protocol by Analyzing Network Data Packets", *Microsoft Systems Journal*, March 1998, pp. 45-63.

10. Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", *Proc. of the 15th ACM Symposium on Operating System Principles*, 1995, pp. 40-53.

11. GigaNet Incorporated, GigaNet cLAN Product Family, http://www.giga-net.com/products

12. Aniruddha S. Gokhale and Douglas C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance", *Proc. of Hawaii International Conference on System Sciences (HICSS)*, January 1998.

13. Java Remote Method Invocation – Distributed Computing for JAVA, http://java.sun.com/marketing/collateral/javarmi.html

14. Mary Kirtland, "The COM+ Programming Model Makes it Easy to Write Components in Any Language", *Microsoft System Journal*, December 1997.

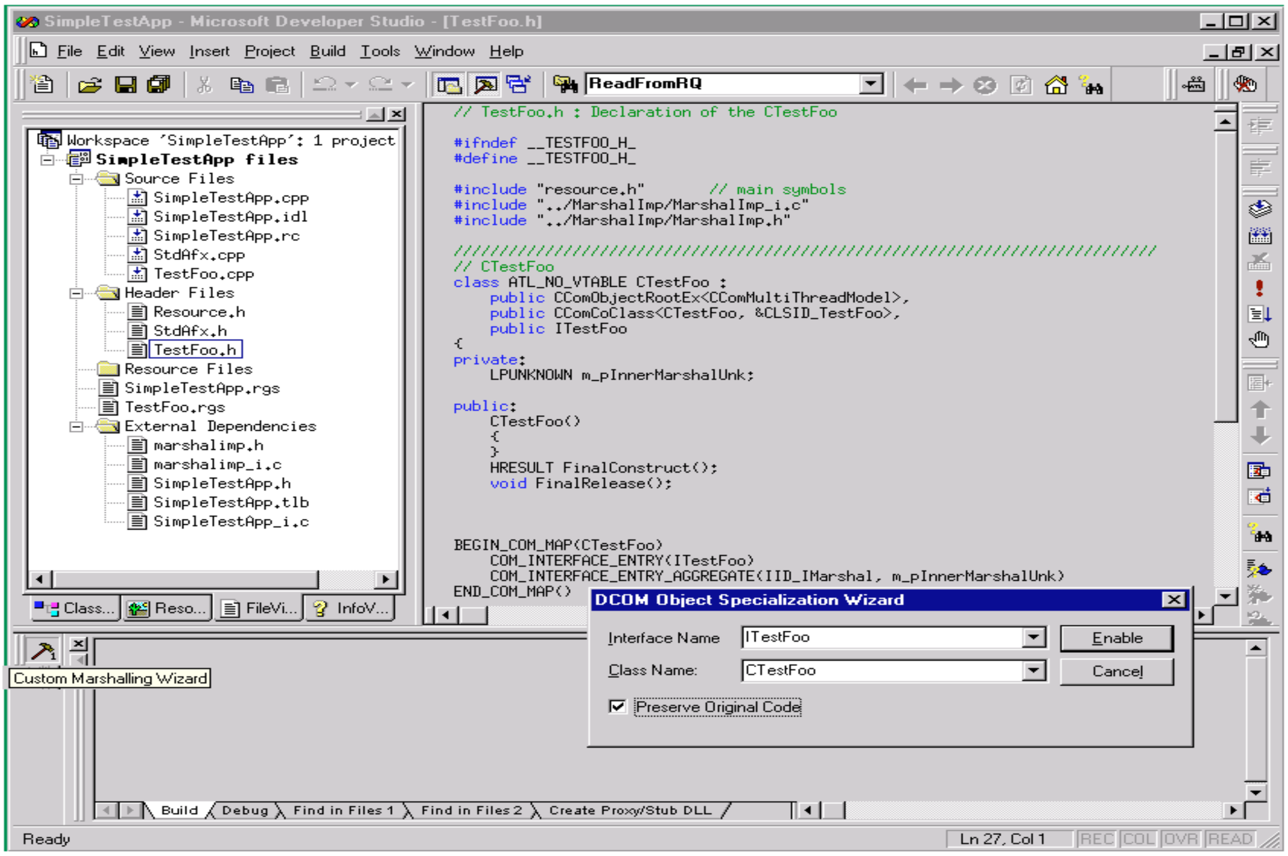
15. Gilles Muller, Renaud Marlet, Eugen-Nicolae Volanschi, Charles Consel, Calton Pu and Ashvin Goel, "Fast, Optimized Sun RPC Using Automatic Program Specialization", *Proc. of the International Conference on Distributed Computing Systems (ICDCS-18)*, May 1998.

16. Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.0 edition, July 1995.

17. Ashish Singhai, Aamod Sane, and Roy H. Campbell, "Quarterware for Middleware", *Proc. of the International Conference on Distributed Computing Systems (ICDCS-18)*, May 1998.

18. Virtual Interface Architecture Specification, Version 1.0, December 1997. http://www.viarch.org/

19. Yi-Min Wang and Woei-Jyh Lee, "COMERA: COM Extensible Remoting Architecture", *Proc. of 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, April 1998. http://www.research.att.com/~ymwang/papers/HTML/COMERA/S.html



```
// TestFoo.h : Declaration of the CTestFoo

#ifndef __TESTFOO_H_
#define __TESTFOO_H_

#include "resource.h"       // main symbols
#include "../MarshalImp/MarshalImp_i.c"
#include "../MarshalImp/MarshalImp.h"

/////////////////////////////////////////////////////////////////////////////
// CTestFoo
class ATL_NO_VTABLE CTestFoo :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<CTestFoo, &CLSID_TestFoo>,
    public ITestFoo
{
private:
    LPUNKNOWN m_pInnerMarshalUnk;

public:
    CTestFoo()
    {
    }
    HRESULT FinalConstruct();
    void FinalRelease();


BEGIN_COM_MAP(CTestFoo)
    COM_INTERFACE_ENTRY(ITestFoo)
    COM_INTERFACE_ENTRY_AGGREGATE(IID_IMarshal, m_pInnerMarshalUnk)
END_COM_MAP()
```

Interface Name: ITestFoo

Class Name: CTestFoo

[✓] Preserve Original Code

Figure 10: A Snapshot of "Custom Marshaling Wizard"