High-Performance Distributed Objects over System Area Networks

Alessandro Forin, *Microsoft Research*

Galen Hunt, *Microsoft Research*

Li Li, *Cornell University*

Yi-Min Wang, *Microsoft Research*

Abstract

In this paper, we describe an approach to build high-performance, commercial distributed object systems over system area networks (SANs) with user-level networking. The specific platforms we use in this study are the Virtual Interface Architecture (VIA) and Microsoft's Distributed Component Object Model (DCOM). We give a detailed functional and performance analysis of DCOM and apply optimizations at several layers to take full advantage of modern high-speed networks. Our optimizations preserve the full set of DCOM features including security, alternative threading models, and Microsoft Transaction Server (MTS). Through extensive runtime, transport and marshaling optimization, our system achieves round-trip latencies of 72 microseconds for DCOM calls and 174 microseconds for MTS calls, and an application bandwidth of 86.1 megabytes per second. We also examine the performance gains in real applications.

1. Introduction

With the explosive growth of the Internet and advances in high-speed networking, distributed computing is becoming a predominant programming paradigm for building mission-critical applications. In recent years, research and development efforts along three lines have significantly changed distributed computing. First, researchers have long observed that the software overhead in the communication protocol stacks accounts for a significant fraction of the end-to-end transmission delay. The intervention of the operating system in the critical path and repeated copying of intermediate buffers hinder the delivery of order-of-magnitude improvements in raw network speed to applications. Several research projects have proposed and implemented fast networking protocols and interfaces that allow user-level access to high-speed networking devices. Examples include Cornell U-Net [V95], Illinois Fast Messages (FM) [P97], Princeton Virtual Memory-Mapped Communication (VMMC) [B94], etc.

Another trend in distributed computing is the increasing popularity of server clusters. By connecting a number of relatively inexpensive machines with high-speed System Area Networks (SANs), such configurations offer a cost-effective approach to achieving high performance and availability. The Virtual Interface Architecture (VIA) [V97], proposed as an industrial standard user-level networking architecture, promises to deliver much of the raw power of SAN directly to applications.

The third trend is object orientation. Distributed object systems, such as Distributed Component Object Model (DCOM) [B98], Common Object Request Broker Architecture (CORBA) [C95], and Java Remote Method Invocation (RMI) [W95], extend the benefits of object-oriented programming to the networked environment. These systems provide an infrastructure that hides the details of low-level communication mechanisms and presents a higher-level abstraction to programmers to simplify distributed programming.

Just as high-speed networks shift the performance bottleneck to protocol stacks, commercial user-level networking shifts the bottleneck to distributed-object infrastructures. In this paper, we use DCOM over VIA as an example to investigate the design issues in providing high-performance distributed object systems over user-level networking. The target application environments are physically secure server clusters consisting of homogeneous machines connected by a high-speed system area network. Our goal is to implement a set of software modules that can be integrated into the existing DCOM infrastructure. These modules will be loaded for inter-server communications within a cluster, while client machines outside the cluster continue to contact the server machines through traditional protocol stacks running over traditional networks.

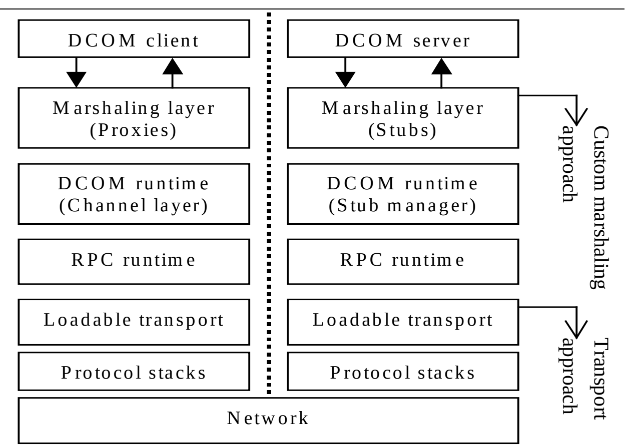

Figure 1. Layered architecture of DCOM.

Figure 1 illustrates the high-level architecture of DCOM. The marshaling layer packs method-call parameters and converts data formats between machines with different architectures. The DCOM runtime maintains object and interface identities, enforces DCOM-level access control, and supports different threading models. The RPC runtime manages thread pools, message multiplexing, interface bindings, and authentication. At the loadable-transport layer, each transport exports a common interface to encapsulate network protocol-specific details from the runtime layer. Loadable transports reside in dynamic link libraries (DLLs) separate from the runtime DLLs.

Figure 1 suggests several approaches to run DCOM over VIA with different tradeoffs. The *custom marshaling approach* [C98][Ma98] uses a custom marshaling layer to run DCOM applications directly on VIA, bypassing all runtime support from DCOM and RPC. This approach can deliver almost all of the raw VIA performance, but does not support the full set of DCOM features. The *transport approach* adds a new loadable transport module that encapsulates all VIA-specific details. The new transport transparently supports all DCOM and RPC applications, however the RPC runtime seriously limits potential performance gains [Z98].

In contrast, our system, *Millennium Falcon*, leverages the existing DCOM runtime layer to support the full set of DCOM features. Optimizations for VIA are implemented in the loadable transport, RPC runtime, and marshaling layers. At the loadable transport layer, we exploit RPC semantics to perform efficient flow control. At the marshaling layer, we achieve zero buffer copies by exposing scatter-gather I/O to proxies and stubs. At the RPC runtime layer, we implement a new binding-handle module to improve DCOM critical-path performance. We address both latency and bandwidth issues. The former is important for the common case of method calls with small-size parameters. The latter is important for applications that perform bulk data transfer, including scientific, database, and checkpoint applications.

The paper is organized as follows. Section 2 gives an overview of Virtual Interface Architecture, RPC, and DCOM. Section 3 presents a layered performance analysis of current DCOM implementation on Windows NT. The design and implementation of Millennium Falcon are described in Section 4, and extensive performance measurements are presented in Section 5. Section 6 covers related work, and Section 7 gives the conclusions.

2. Background

2.1. Virtual Interface Architecture (VIA)

The Virtual Interface (VI) Architecture is an industrial, user-level networking architecture for high-bandwidth, low-latency, and low-overhead communication [V97]. In contrast to traditional network architectures where the operating system virtualizes network hardware into a set of logical endpoints, the network adapters in the VI architecture take over the task of endpoint virtualization, data multiplexing and transfer scheduling to reduce OS involvement. Each VI represents an endpoint with a direct, protected interface to network hardware. A process may acquire multiple VIs exported by one or more network adapters.

A *VI consumer* is typically an application program that uses VIs through some communication facility such as sockets. The communication facility usually loads a *VI user agent* library supplied by the hardware vendor to abstract the details of the underlying VI provider. The *VI provider* consists of a network interface controller that implements the VIs and directly performs data transfer functions, and a *VI kernel agent* that performs setup and resource management functions. The VI consumer registers the memory buffer with the VI provider before submitting a data transfer request. This allows the consumer to reuse the registered memory buffer for subsequent data transfers to avoid the overhead of duplicate page lock/unlock operations and address translations.

Each VI consists of two *work queues*: a *send queue* and a *receive queue*. A request from the VI consumer to send (or receive) data is posted on the send (or receive) queue as a *descriptor*, which contains all the information needed by the VI provider to process the request. A *doorbell* associated with each work queue notifies the network adapter that a new descriptor has been posted. The VI provider asynchronously processes the descriptors and marks them with status values upon completion.

2.2. Remote Procedure Call (RPC) and Distributed Component Object Model (DCOM)

Microsoft RPC is an implementation of the DCE RPC [D95] specification. It uses the Network Data Representation (NDR) format for data marshaling in heterogeneous environments. Typical RPC client and server applications are structured as follows. The server application specifies a protocol in an RPC API call, which loads the corresponding transport DLL and creates a communication endpoint. The server invokes another API to register the RPC interfaces on which it expects to receive method calls. An *interface* consists of a set of functionally related method calls. Each RPC interface is identified by a 128-bit Universally Unique Identifier (UUID) called an *Interface ID* (or *IID*), and is specified in an Interface Definition Language (IDL) file. Once the server is ready to receive calls, it constructs an RPC *string binding*, which contains sufficient information to identify the server endpoint. The string binding is propagated to the client through a naming service or some other means. The client constructs a binding handle from the string and makes RPC calls through the handle.

DCOM extends DCE RPC to support the notion of objects with multiple interfaces and to provide a mechanism for server activation. At the marshaling layer, the current DCOM implementation reuses MSRPC NDR code for data marshaling and augments it with the support for marshaling *object interface pointers* into object references. An *object reference* contains sufficient information to locate a unique object interface instance within a unique server endpoint. On the client side, DCOM inserts a *channel layer* between the marshaling layer and RPC runtime, as illustrated in Figure 1. Each channel object encapsulates an RPC binding handle and manipulates the DCOM-specific part of each RPC packet. The counterpart of the channel object on the server side is the *stub manager*, whose main task is to dispatch each incoming call to its target stub.

Typical DCOM client-server interactions proceed as follows. The client application invokes the *CoCreate-InstanceEx()* API to either activate a server application or connect to a running server process. The client specifies a *Class ID* (or *CLSID*), which is the UUID of an object class, and the IID of the interface to which it is requesting a pointer. As a result of this activation, the server process creates an object instance of CLSID and (logically) returns to the client a pointer to the object's IID interface. The client can then invoke methods through that pointer as if the object resides in the client's own address space. When the client needs a pointer to another interface of the same object instance, it makes a *QueryInterface()* call on the current interface pointer and supplies a new IID. In Section 4, we will describe in more detail how current DCOM implementation provides this object-oriented abstraction on top of MSRPC, and discuss its impact on performance.

3. Where Do The Cycles Go

We present here an analysis of the overhead distribution among layers in the current DCOM implementation over TCP. Measurements were obtained by analyzing DCOM/RPC source code and intercepting layer-crossing calls through binary instrumentation. The analysis identifies performance bottlenecks and predicts the effectiveness of candidate optimizations. Our measurement setup consists of a pair of Gateway 2000 E-5000 PCs, each with a 333 MHz Pentium II CPU and 256 Mbytes of RAM, and running Windows NT Workstation 4.0 with Service Pack 3 (NT4 SP3). The machines are directly connected through a pair of 1.25-Gb/s GigaNet GNN 1000 adapters. The test program, *PingPong*, invokes a DCOM call to send buffers of the same size back and forth. The buffer size is specified as the first method parameter followed by the pointer to the buffer specified as an `[in,out]` parameter. The standard, compiled proxy/stub code generated by the Microsoft IDL (MIDL) compiler is used.

Figure 2 illustrates the round-trip overhead distribution across different DCOM data sizes. All numbers are averaged over 1,000 runs. In both plots, at the top layer, the marshaling overhead results from gathering call parameter data into the RPC buffer. In the middle, the runtime overhead represents the data-size-independent portion of DCOM and RPC processing. At the bottom, the transport overhead includes the execution times of the RPC loadable transport, the TCP protocol stack, and the actual wire time.

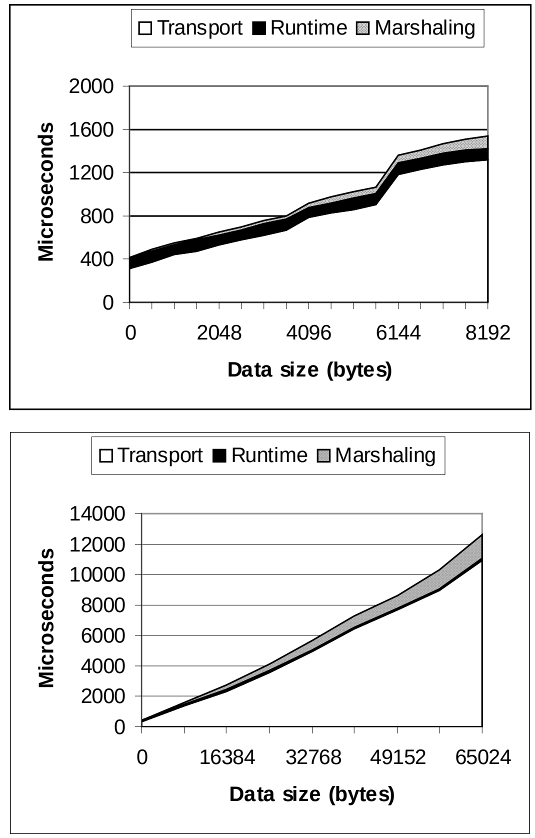

Figure 2. Round-trip overhead distribution of DCOM over TCP: small data size (top); large data size (bottom).

Round-trip Latency (null call)	413 µs
Round-trip Latency (64KB)	12.6 ms
Max. DCOM Bandwidth	11.4 MB/s
DCOM/RPC Runtime	100 µs
Marshaling Time	7 µs + 18 µs/KB + 34 µs/KB above 50K
Transport Time	313 µs + 110 µs/KB + 260 µs/5840 bytes

Table 1. Summary of DCOM-over-TCP performance numbers on GigaNet GNN 1000.

Table 1 summarizes the essential numbers from the two plots. The average round-trip latency for DCOM calls without any parameters is 413 µs. At the other extreme, the latency for close to 64KBytes is 12.6 ms. (The Maximum Transfer Unit, MTU, for the GigaNet cards is 17 bytes below 64KB.) The maximum bandwidth of 11.4 megabytes per second is achieved at a data size around

20KB. The fixed DCOM and RPC runtime overhead is approximately 100 µs.

Table 1 also gives the analytical equations for the curves that fit the transport time and marshaling time. Since the RPC-over-TCP implementation uses a maximum buffer size of 5840 bytes, there is an additional fixed overhead for every 5840 bytes (the discontinuity in the top graph of Figure 2).

Both plots show that transport overhead dominates round-trip latency for the simple *PingPong* program, which suggests that replacing the slow kernel-mode protocol stack with a fast user-mode network such as VIA can significantly improve DCOM application performance. Specifically, the transport overhead accounts for 313 µs (76%) of the 413 µs round-trip latency for the null-call case, and is responsible for 11 ms (87%) of the 12.6 ms at the other extreme. With VIA, the transport times can be reduced to 30 µs and 1.4 ms, respectively.

The results also imply that once the transport overhead is reduced, runtime and marshaling overhead will become significant. Specifically, at the low end, DCOM applications would be ((100+30)-30)/30 = 333% slower than raw VIA applications. At the high end, DCOM applications would be ((12.6-11+1.4)-1.4)/1.4 = 114% slower than raw VIA applications and marshaling overhead would account for (12.6-11-0.1)/(12.6-11+1.4)=50% of latency, which translates into a 50% reduction in application bandwidth, compared with raw VIA. To make DCOM a competitive programming environment in the high-performance application market, we must apply optimizations at the runtime and marshaling layers to reduce overhead.

4. Millennium Falcon: Fast DCOM over VIA

Motivated by the overhead analysis described in the previous section, our Millennium Falcon prototype applies optimization at three layers of the DCOM architecture. At the transport layer, we take advantage of RPC semantics to perform efficient flow control. At the marshaling layer, we provide support for marshaling and unmarshaling of DCOM application buffers directly to and from network adapters. At the runtime layer, we analyze the critical paths of DCOM calls and build a new RPC binding-handle module to eliminate unnecessary RPC runtime overhead for DCOM applications. The effectiveness of the transport-layer optimization is specific to VIA, while the other two optimizations are applicable to the existing DCOM/RPC protocol stacks as well. We implemented the optimizations by modifying the NT4 SP3 DCOM and RPC source code and adding new modules for the new stack.

4.1. Transport and Marshaling Optimization for User-Level Networking

4.1.1. RPC Loadable Transport

To build a new RPC loadable transport for user-level networking, we first implemented a socket layer over VIA to hide the details of memory registration, VI management, etc. We then built the VIA loadable transport module on top of the socket layer to make it portable across different hardware and VIA implementations.

To preserve performance in a layered architecture, special attention must be paid to flow control. In VIA, incoming data is delivered directly, through DMA, to user-level buffers without kernel buffering. The receiver must post its buffers before the sender commences transfer; otherwise, VIA fails the transmission and closes the connection. For reliable communication, VIA applications must employ active flow control. Our first choice was to implement flow control in the socket layer. However, preliminary measurements showed that socket-layer flow control adds about 130 µs to the round-trip time, severely restricting performance. This overhead is mainly due to context switches and additional interrupts on both sides.

While socket-layer flow control is necessary for arbitrary applications, more efficient flow control can be achieved at a higher level by exploiting the RPC semantics. To achieve this, our socket layer supports optional disabling of flow control. Our loadable transport performs RPC-specific flow control as follows. When the server-side transport is about to accept a connection request from a client, it pre-posts a buffer for receiving the first method call so that the client can make calls immediately after it successfully makes a connection to the server. Before the client-side transport sends out the marshaled method call, it pre-posts a buffer for receiving the reply so that the server is free to send back a reply any time after the request is processed. Similarly, before the server sends a reply, it pre-posts a buffer for receiving the next request from the client. Essentially, in the RPC context, flow control messages are piggybacked on request and reply messages. Such an optimization is compatible with the original implementation in which concurrent calls between threads of the same pair of processes do not share the same socket connection. With this optimization, round-trip latency is reduced to 30 µs for small data sizes, an order-of-magnitude improvement over the transport overhead for DCOM over TCP (313 µs).

Two issues remain beyond the above simple flow control. First, VIA transmission will fail if the receiver buffer is smaller than the incoming data packet. In our current prototype, both the client and the server post buffers with size equal to the MTU of the network, which is close to 64KB on GigaNet GNN1000. A more practical solution is to adopt a default size smaller than the MTU and sufficient for most cases. When one side occasionally needs to send data larger than the buffer size, it first sends a control message (with 30-μs round-trip time) to the other side to request a larger buffer, and then sends the actual data. Alternatively, if the network adapters support true Remote Direct Memory Access (RDMA) mode, the control message can request a large RDMA buffer on the receiving side and the actual data can be transferred using one VIA descriptor.

Second, even in the RPC context, there can be non-RPC-style communication. For example, as will be discussed later, Millennium Falcon uses Windows NT LanManager challenge-response protocol for connect-time authentication. This protocol consists of three legs: client sends an authentication request, server replies with a challenge, and finally client sends a response. At this point, the client is free to send its first request if using a traditional kernel-mode network protocol. In our prototype, special care needs to be taken because we rely on the RPC semantics for flow control. A straightforward solution is to add a fourth leg from the server back to the client to maintain the RPC semantics. Alternatively, the third leg can be combined with the first client call.

4.1.2. Eliminating Data Copying

While user-level networking eliminates data copy between user-level buffers and kernel buffers, the data marshaling process in DCOM and RPC introduces another data copy at a higher layer. Specifically, the proxy code is responsible for gathering method call parameters and packing them into an RPC buffer. This buffer can be directly accessed by the network adapters without additional copying. The analysis in Section 3 showed that the remaining data copy at the marshaling layer could significantly limit the achievable application bandwidth. Intuitively, because VIA supports scatter-gather operations and data conversion is not needed within a homogeneous cluster, it is possible to have network adapters directly access the memory regions of each individual call parameter without an intermediate copy. One could imagine a general scheme in which the IDL compiler is modified to generate proxy/stub code that does not copy call parameters into the RPC buffers; instead, the code constructs a gather or scatter list for VIA DMA access.

While such a general solution to optimal marshaling performance in user-level networking is beyond the scope of this paper, our current prototype provides support to reduce data copying on both the client and the server sides. Applications that require high bandwidth can supply modified proxy/stub code to take advantage of the support. We now describe our marshaling optimization by following the sequence of steps of a method call that involves only simple arrays of basic types.

Client side sending a request. A new RPC flag is added to allow modified proxy/stub code to declare that it is requesting the scatter-gather mode of transmission. When this flag value is set, the RPC runtime interprets the data buffer as a list of scatter-gather entries, each consisting of a starting memory address and a data length. The RPC runtime adjusts the entries to include RPC headers and passes the list down to the loadable transport, which calls the socket layer to perform a gather-send operation. Unflagged proxies and stubs still use copy-mode marshaling.

Server side receiving a request. Since the server-side RPC runtime may receive calls on any methods supported by the server process, it is in general not possible to specify a receive scatter list for any arbitrary method in advance. However, since the receiving RPC buffer is dedicated to the on-going RPC call for its entire duration, the stub code and the method implementation can use the buffer data directly without copying. Even for complex data structures, intelligent proxy/stub code can adjust data layout to use the incoming buffer directly [C98].

Server side sending a reply. This is basically similar to the client-side send. The only complication is when the client passes a pointer as an [out] parameter to request a variable-size array, whose size is determined at run time by the server. In this case, a properly designed COM interface method allocates space for the resulting array by calling *CoTaskMemAlloc()*, and the client is responsible for calling *CoTaskMemFree()* to free the buffer. To support this memory allocation paradigm, the original proxy/stub code does the following: the server-side stub frees the server-side application buffer once it has been copied into the reply RPC buffer; the client-side proxy allocates an application buffer once it knows the proper size, and copies the data from the RPC buffer to the application buffer.

On a user-mode network, the stub code cannot free the application buffer until the loadable transport has finished sending out the entire data directly out of the buffer. To simplify the task of modifying the standard proxy/stub code to delay freeing the buffer, we allow the stub to pass down a *callback function pointer* and a *context pointer*, along with the scatter-gather list. When the loadable transport finishes using the application buffer, it invokes the callback function with the context pointer. The callback function frees the application buffer.

Client side receiving a reply: Unlike the server side, since the client-side receive operation expects a reply for a particular method call, the proxy code can specify a reply scatter list to transfer incoming data directly into the memory of the [out] parameters and the return value. However, the variable-size array example mentioned previously poses certain restrictions. First, the size of the return array is unknown to the proxy when the call is made. It is not possible for the proxy to pre-allocate a buffer of the right size and post it in a scatter list. The reply must be received by an RPC buffer in this case. Furthermore, unlike the server side, it is not desirable for the client application to use the data directly from the RPC buffer because it may hold on to the buffer for an undeterminable period before calling *CoTaskMemFree()*. So a memory copy is still needed in this case.

4.2. RPC Runtime Optimization for DCOM

Once the transport and marshaling optimizations are in place, the fixed runtime overhead of 100 μs becomes highly visible. In this section, we investigate how RPC runtime can be optimized to efficiently support DCOM. Optimizations at this layer are generally applicable to speeding up DCOM on any network, although the performance gain is much more significant in the SAN environment. As briefly introduced in Section 2, DCOM was designed as a thin layer on top of DCE RPC to leverage existing distributed system functionality provided by RPC. We re-examine this design decision from a performance point of view. Our study shows that, while DCOM benefits from the threading, connection, and security management support of RPC, it suffers from the unnecessary performance penalty due to RPC's interface management. We then describe the design and implementation of a new RPC runtime module that is optimized for DCOM.

4.2.1. Critical Path Analysis

Distributed object systems such as DCOM provide an abstraction to simplify distributed programming by hiding low-level communication details. To identify critical paths and apply effective optimization, we must understand how network traffic is generated in response to DCOM calls. The following is a critical-path analysis of critical events for typical DCOM client-server interactions.

Getting the first interface pointer to an object. When a DCOM client invokes *CoCreateInstanceEx()*, the client-side Service Control Manager (SCM) forwards the request to the SCM on the server machine. The server-side SCM locates or starts a process capable of hosting objects of the requested class. When the object implementation returns a pointer to the requested interface, DCOM marshals that pointer into an object reference by generating a 128-bit locally unique *Interface Pointer Identifier* (*IPID*) that identifies that particular interface instance. DCOM also registers with the RPC runtime the requested interface ID (IID) and an access control callback function if support for security is desired. Finally, the SCM asks the server to create an endpoint and construct a string binding. The SCMs then ferry both the object reference and the string binding back to the client process. Together the object reference and string binding uniquely identify the newly created interface instance. The client-side DCOM runtime constructs a binding handle from the string, loads the appropriate proxy code for the requested IID, and returns to the client a pointer to a proxy interface. Note that at this point no socket connection has been established between the client process and the server process.

We do not consider performance optimization for *CoCreateInstanceEx()* in this paper for two reasons. First, *CoCreateInstanceEx()* is only called once when the client initiates the first contact with an object. For applications that make a large number of subsequent method calls, object creation can be considered out of the critical path. Second, since *CoCreateInstanceEx()* involves expensive object creation logic, its performance will not be significantly improved by low-level runtime and transport optimizations alone. Changing the application architecture to move *CoCreateInstanceEx()* out of the critical path would be a much more effective solution. For example, server process *A* can pre-fetch and cache interface pointers from server process *B* running on another node of the same SAN cluster.

Making the first call to a server process. When the client makes the first call on a newly acquired interface pointer, the RPC runtime tries to reuse an existing socket connection to the server process that hosts the target object. If none is available, it opens a new socket connection. After the server accepts the request, if the authentication mode is turned on, the client starts the authentication process by sending a BIND message. Different authentication protocols may require different numbers of security related messages to be exchanged. As the result of a successful authentication, *security contexts* representing the calling principal are established on both sides. Finally, the client-side RPC runtime sends the request on the newly opened connection. Since the overhead of this entire connection process is amortized across all method calls between the client-server pair, it can be considered out of the critical path. If desirable, a process can pre-fetch an interface pointer from another process and make one initial call on it to set up the connection and security contexts in advance.

Making the first call to an interface. As part of the BIND process for establishing security contexts, a *presentation context* associated with the IID of the call is also established. A per-IID binding entry is inserted into the binding dictionary on both sides. The purpose of the presentation context is to ensure that the server indeed supports the requested IID, and to facilitate future call dispatching on the server side, which basically involves mapping an IID to the dispatch function and security callback function that were registered for it. When the DCOM client makes the first call to another IID of the same process, the RPC runtime sends an *alter-context* round trip to establish a presentation context for the new IID. In contrast, the socket connection and its associated security contexts are shared across IIDs.

Since it is quite common for DCOM clients to request an interface pointer and make only a very small number of calls on that pointer, the first calls to each IID should be considered inside the critical path. The extra alter-context round-trips may impose serious performance degradation. From DCOM's point of view, these round trips are needless overhead for three reasons. First, when a DCOM client successfully obtains an interface pointer through either the *QueryInterface()* or *CoCreateInstanceEx()* calls, it has the implicit acknowledgement from the server that the IID is supported and has been properly registered. The additional RPC-level verification is redundant. Second, because all DCOM interfaces register the same RPC dispatch function, namely the entry point to the DCOM stub manager, it is unnecessary for the RPC runtime to lookup the binding dictionary to map an IID to its dispatch function. (The true DCOM call dispatching is performed inside the stub manager based on IPID.) Third, since all DCOM interfaces register the same per-process RPC access control function, the RPC runtime IID to security callback mapping is not needed.

Based upon the above analysis, we conclude that the interface (IID) management at the RPC runtime layer is unnecessary for DCOM applications. In our new binding-handle module optimized to run DCOM, we removed the entire IID notion from this layer to eliminate unnecessary network traffic, memory consumption, and performance overhead.

Making additional calls. The rest of the calls are certainly on the critical path. Removing the notion of IID from the RPC runtime layer improves the performance at several places along the critical path. On the client side, the binding dictionary lookup is eliminated. Since this operation required locking a dictionary shared by multiple threads, the performance gains are even larger for multi-threaded clients. The RPC runtime need not match presentation context to find a connection with the correct security context. On the wire, the IID is not transmitted because the IPID alone is sufficient for correct dispatching. On the server side, the RPC runtime invokes the security callback function and the DCOM stub manager directly without any dictionary or table lookup. Latency numbers in Section 5 demonstrate the effectiveness of these optimizations.

4.2.2. Binding-handle Optimization

We implemented our RPC runtime optimization as a new binding-handle module. This module sits under the common RPC API layer, which provides handle-type-independent processing. Our new module operates at the same level as the three existing binding-handle modules for connection-oriented protocols, connectionless protocols, and Local Procedure Calls. On the client side, when a call is made on a handle instance, the module finds the target-server endpoint. Then it searches for an available socket connection for the endpoint and, if there is none, creates a new connection. On the server side, the module maintains a thread pool based on the total number of active calls. Each thread, upon receiving a client request, performs appropriate security checks and forwards the request to the DCOM stub manager.

Security is one of the most important features provided by DCOM and RPC, and it plays an essential role in commercial client-server applications. DCOM security consists of three parts: *authentication* for verifying clients' identities and messages' authenticity; *access control* for restricting object access rights to a subset of users; and *impersonation* for allowing servers to execute under clients' credentials. Our new module supports all three security functions, while taking into account the special characteristics of the SAN environments.

DCE RPC supports multiple levels of authentication including *connect-time-only authentication*, *per-call or per-packet header protection*, *per-packet payload protection,* and *per-packet encryption*. Because our target environment is a physically secure server cluster, the last three protections against malicious attacks from outside agents are unnecessary. However, connect-time authentication is still necessary to prevent one legal user from gaining unauthorized access to the private data of another legal user. Specifically, once a client obtains an interface pointer, almost all future interactions between the client and the server will happen directly between the DCOM and RPC runtime libraries running inside the application processes. Therefore, even if the machines and kernels within the cluster trust each other, DCOM applications do not necessarily trust each other and so support for connect-time authentication is necessary. Fortunately, the overhead of this level of security is mostly outside the critical path, as discussed previously.

Our current prototype uses Windows NT LanManager challenge-response protocol for connect-time authentication. Once the client is authenticated, security contexts established on both sides serve as the basis for other security functions. When the server invokes DCOM APIs to impersonate the client, the task is mostly delegated to the security context. When the DCOM runtime performs an access permission check, it first impersonates the client and then retrieves the thread token to check against the access control list. Since DCOM provides APIs for client applications to dynamically change security-related information on a per-call basis, the client-side binding-handle module ensures that every method call is sent along a socket connection with the correct security context.

5. Performance Measurements

In this section, we present the performance comparison between our Millennium Falcon prototype and the existing DCOM implementation to quantify the performance gains of our optimizations. We use the same hardware setup as described in Section 3 for most of the measurements.

5.1. Round-Trip Latency

Figure 3 compares the round-trip latency for data sizes ranging from 0 to 8K bytes. The curves marked *VIA-copy* and *VIA-direct* represent DCOM over VIA with and without RPC buffer copying, respectively, as described in Section 4.1.2. For data of size zero, our implementation reduces the round-trip latency from 413 μs to 74 μs, a more than 5-time improvement. (The best number is 72 μs, with 74 μs being the average over 1,000 runs.)

Profiling data indicate that the overhead distribution is approximately 42 µs for runtime and 32 µs for transport, compared to 100 µs and 313 µs in the TCP case. For data of size 8K bytes, VIA-copy reduces the latency from 1540 µs to 457 µs and VIA-direct further reduces the number to 268 µs. The analytical equation for the VIA-direct curve is 22 µs (marshaling) + 42 µs (runtime) + (32 µs + 21 µs/KB * size) (transport). (The marshaling overhead as shown here is independent of the data size, but it can be a function of the number of call parameters in general.)

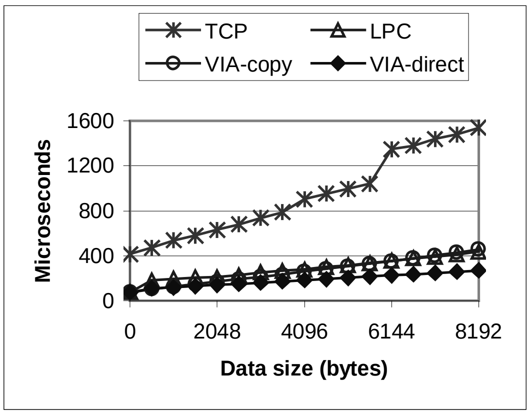

Figure 3. Comparison of DCOM round-trip latency (*PingPong*) over TCP, LPC, and VIA with and without data copying.

We also include the curve for DCOM over *Local Procedure Calls (LPCs)* in Figure 3. (LPC is one of the cross-process communication mechanisms within a single Windows NT machine.) The figure shows the interesting fact that, for this data range, cross-process local DCOM calls and cross-machine remote DCOM calls over VIA (with buffer copying) actually have very similar performance. The null-call round-trip latency for the DCOM over LPC case is 73 µs, where 45 µs comes from runtime and 28 µs from LPC. By eliminating data copying, the latency of VIA-direct at data size 8K is only about 60% of the corresponding DCOM-over-LPC latency. This result can have great impact on system designs that must decide between local versus remote execution and communication. For example, algorithms that tend to place communicating processes on the same machine in order to minimize communication overhead now have the flexibility of spreading processes across the network to take advantage of remote processing power.

For Figure 3, both client and server threads spin waiting for an incoming message to avoid delays due to context switching. For a discussion on the performance impact of blocking, see the full technical report [L98].

5.2. Application Bandwidth

Figure 4 compares DCOM application bandwidth for the three implementations. Due to the high overhead of the kernel-mode protocol stack, inefficient transport management assuming small network MTUs, and several intermediate buffer copies, the current DCOM-over-TCP implementation can only deliver a maximum bandwidth of 11.4 MB/s. Using user-level networking and taking advantage of the large MTU, VIA-copy increases the maximum bandwidth to 43.4 MB/s. However, that is still less than 50% of the raw VIA bandwidth of the GigaNet GNN1000 (about 105 MB/s). By eliminating all buffer copies on both sides, VIA-direct achieves a bandwidth of 86.1 MB/s.

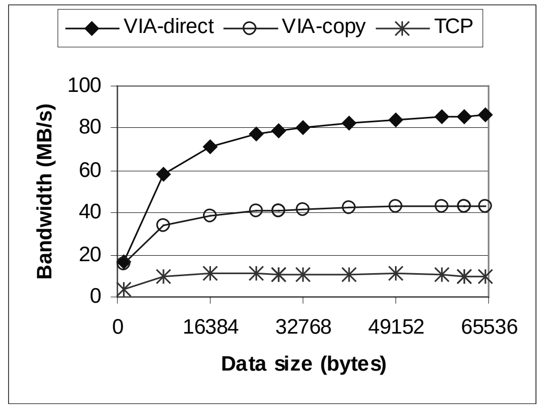

Figure 4. Comparison of DCOM application bandwidth (*PingPong*) over TCP and VIA.

The dramatic increase in available bandwidth demonstrates the importance of the marshaling layer optimization. Today, many applications use DCOM as their main programming paradigm, but use separate, lower-level communication primitives such as sockets for bulk data transfer. Since the VIA-direct implementation preserves 96% of the bandwidth available to a raw socket implementation (90.1 MB/s), a consistent programming paradigm based on DCOM can now be applied throughout an application for both control and bulk data transfer.

5.3. Secure Calls and First Interface Calls

Next, we compare the round-trip latency of secure calls and the first call to a new IID (after a socket connection has been established). The top portion of Table 2 shows that, with security turned off, the latency of the first call to an IID is more than twice that of later calls in the TCP case. Overhead comes from the alter-context round trip (Section 4.2.1), the initialization of the RPC binding handle, and the initialization of the DCOM channel object. By eliminating the alter-context round trip and by moving the initialization of RPC binding handle out of the critical path, our implementation achieves a round-trip latency of 109 µs, which is 8 times faster than its TCP counterpart. The remaining overhead of 109-74=35 µs is due to per-interface DCOM channel initialization for supporting per-interface security.

The lower portion of Table 2 shows latency when connect-time authentication is enforced. Ideally, this level of security should impose zero overhead for all calls using an existing connection. However, the current DCOM-over-TCP implementation has a 448-413 = 35 µs overhead that comes from two sources. First, the client-side RPC runtime must match, on a per-call basis, the security setting of the current call against the security context of existing connections. The second source of overhead is an implementation issue where the RPC layer performs security tasks that are unnecessary for DCOM. By removing the second source in our binding-handle module, the per-call overhead is reduced to only 76-74 = 2 µs.

Without Security	**Later Calls**	**First Calls**
DCOM over TCP	413 µs	880 µs
DCOM over VIA	74 µs	109 µs
With Security (Connect-time-only)	**Later Calls**	**First Calls**
DCOM over TCP	448 µs	1600 µs
DCOM over VIA	76 µs	300 µs

Table 2. Comparison of DCOM round-trip latency for secure and non-secure, first and subsequent interface calls.

The lower portion of Table 2 shows that, for DCOM over TCP, the latency of first secure calls is more than 3 times that of later calls. A detailed discussion on the source of the overhead and our optimizations can be found in [L98]. For the VIA case, out of the 300 µs, about 186 µs are spent on acquiring the caller's credentials for supporting dynamic security.

5.4. Apartment Threading

DCOM supports two different threading models for servers: free threading and apartment threading. So far, we have limited our discussion to the free threading model, in which the RPC threads responsible for receiving client requests are also the threads that eventually invoke the object stubs. In the free threading model, the application programmer must synchronize data accesses. In contrast, apartment threading simplifies concurrent programming by providing automatic access synchronization. By having a single thread operate in a Single-Threaded Apartment (STA), all accesses to objects in that STA are serialized and there is no need for additional programmatic synchronization.

The servers-side DCOM runtime implements apartment threading by posting each incoming DCOM call to the Windows message queue associated with the target STA, thereby serializing all calls into that STA. Apartment-threaded servers are in general slower than free-threaded servers due to the context switches between RPC and application threads. Unlike transports based on custom marshaling, Millennium Falcon preserves full support for apartment threading.

Our measurements show that, for both TCP and VIA, apartment threading adds an additional 90 to 160 µs on top of the free-threading numbers across different data sizes. (See [L98] for performance graphs.) In particular, the null-call latency in the VIA case more than doubles to 168 µs, and the maximum bandwidth is reduced by 7% to 80.3 MB/s. This suggests that the current implementation of apartment threading may need to be redesigned to take full advantage of user-level networking.

5.5. Microsoft Transaction Server

Microsoft Transaction Server (MTS) provides a runtime environment combining the features of a Transaction Processing (TP) monitor and an object request broker. MTS supports the notion of automatic transactions: a developer builds a COM object in DLL form and registers it with MTS. When a client activates the object, an MTS surrogate process loads the DLL and automatically wraps it with a transaction context so that the object can participate in transactional interactions. MTS is currently built on top of apartment threading.

Table 3 compares the latency for null MTS calls. For the VIA case, of the 280 µs for an MTS call with role-based security, 168 µs are from apartment threading, 6 µs from transactional wrapping, 2 µs from DCOM security, and 104 µs from MTS security. More efficient implementations of apartment threading and MTS security are needed once the DCOM and RPC layers have been optimized for user-level networking.

	Without Security	With MTS Security
MTS over TCP	558 µs	698 µs
MTS over VIA	174 µs	280 µs

Table 3: Round-trip latency comparison for MTS calls.

5.6. Real Applications

We next discuss two categories of DCOM applications and present some preliminary results. The first category is *off-the-shelf consumer applications*. We use the distributed version of Microsoft *PhotoDraw 2000*, created by the Coign auto-partitioning tool [H99] to run on two machines. *PhotoDraw 2000* is an application for manipulating digital images. It is composed of approximately 112 COM component classes in 1.8 million lines of C++ source code. In the particular runs used in our measurements, *PhotoDraw 2000* loaded a 3MByte graphical composition from storage, displayed the image, and exited. It created 295 COM objects and made approximately 9000 DCOM calls. Results showed that Millennium Falcon reduced the total execution time from 24.0 seconds to 21.8 seconds, a 9.2% improvement in performance. On average, the per-call performance gain is around (24.0 - 21.8) / 9000 = 244 µs.

The second category is *client/server applications involving database operations*. Our measurement shows that an MTS-over-TCP call involving an SQL SELECT operation takes 3.45 ms on SQL Server 6.5. Making the same call over Millennium Falcon reduces the round-trip latency to 3.14 ms, a 9.0% performance gain. For an SQL

INSERT operation, the corresponding numbers are 5.06ms vs. 4.79ms, a 5.3% improvement.

6. Related Work

Madukkarumukumana *et al*. built a custom marshaling layer for DCOM over VIA [Ma98]. Their implementation sacrifices DCOM features for speed. Zimmer and Chien built a UDP loadable transport for MSRPC over Illinois Fast Messages [Z98]. They pointed out that the current RPC implementation imposes serious limitations on potential performance gains. Our work on RPC runtime optimization was partly motivated by their observations.

Bilas and Felten [B97] modified SunRPC to run over Shrimp VMMC. In their SunRPC-compatible implementation, their marshaling layer optimization and ours share similar basic ideas, but differ in IDL semantics. Their second system, ShrimpRPC, forgoes application compatibility with SunRPC to allow further optimizations. In contrast, our goal was an efficient RPC layer capable of supporting existing DCOM applications.

Gokhale and Schmidt [G96][G97] evaluated commercial CORBA and RPC implementations. They discovered that some had inefficient server-side dispatching, and that the conversion of typed data to XDR format in SunRPC is a major source of extraneous overhead for homogenous machines. Unlike the studied CORBA implementations, DCOM's dispatch mechanism is already highly optimized, and the NDR conversion in MSRPC occurs only at the receiver side and only if sender and receiver employ different representation formats.

Two classes of optimization have been proposed for the marshaling layer. The first class reduces data copying. Thekkath and Levy [T93] marshaled RPC arguments directly in the kernel to avoid data copying to kernel buffers for ATM and FDDI. This optimization is unnecessary for user-level networking. Some of our optimizations are similar to the buffer caching and aggregation used in [D93], although they dealt with additional cross-domain issues. The second class of marshaling optimizations applies known compiler transformations to stub generation [E97][Mu98]. Such optimizations are essentially orthogonal to our work.

7. Conclusions

We have demonstrated an effective approach to reducing round-trip latencies and increasing application bandwidth for a commercial distributed-object system over user-level networking. Just as high-speed networks shifted the performance bottleneck to the protocol stacks and user-level networking shifted the bottleneck to the communication infrastructures of distributed object systems, our optimizations have again shifted the bottleneck to the support for security and threading, and the initialization overhead of internal data structures. Performance measurements suggest that existing architectures and implementations in these areas need to be carefully reevaluated in order to take full advantage of high-speed networking.

Acknowledgement

We thank Jim Gray, Karin Petersen, and the anonymous reviewers for their valuable comments.

References

[B97] A. Bilas and E. W. Felten, "Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface," in *J. Parallel and Distributed Computing*, Feb. 1997.

[B94] M. A. Blumrich et al., "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," in *Proc. Int. Symp. on Computer Architecture*, pp. 142-153, 1994.

[B98] N. Brown and C. Kindel, Distributed Component Object Model Protocol -- DCOM/1.0, 1998.

[C95] The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995.

[C98] C.-C. Chang and T. von Eicken, "A Software Architecture for Zero-Copy RPC in Java," Cornell CS Technical Report 98-1708, Sep. 1998.

[D95] DCE 1.1: Remote Procedure Call Specification.

[D93] P. Druschel and L. L. Peterson, "Fbufs: A High-bandwidth Cross-domain Transfer Facility," in *Proc. SOSP, 1993*.

[E97] E. Eide et al., "Flick: A flexible, optimizing IDL compiler," in *Proc. ACM SIGPLAN'97 Conf. On Programming Language Design and Implementation (PLDI)*, pp. 44-56, June 1997.

[G96] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-speed Networks," in *Proc. SIGCOMM*, Aug. 1996.

[G97] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *IEEE Trans. on Computers*, Vol. 47, No. 4, 1998.

[H99] G. Hunt and M. L. Scott, "The Coign Automatic Distributed Partitioning System," in *Proc. OSDI*, 1999.

[L98] L. Li et al., "High-Performance Distributed Objects over a System Area Network," Tech. Rep. MSR-TR-98-68, Microsoft Research, 1998.

[Ma98] R. S. Madukkarumukumana, C. Pu, and H. V. Shah, "Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks," in *Proc. USENIX NT Symposium*, pp. 127-135, Aug. 1998.

[Mu98] G. Muller et al., "Fast optimized Sun RPC using automatic program specialization," *Proc. ICDCS, May 1998*.

[P97] S. Pakin, V. Karamcheti, and A. A. Chien, "Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs," *IEEE Concurrency*, 5(2):60-73, 1997.

[T93] C. A. Thekkath and H. M. Levy, "Limits to Low-latency Communication on High-speed Networks," *ACM Trans. on Computer Systems*, 11(2):179-203, 1993.

[V97] Virtual Interface Architecture Specification, Version 1.0, Dec. 1997. (http://www.viarch.org)

[V95] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proc. ACM SOSP, 1995*.

[W95] A. Wollrath, R. Riggs and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Journal, Computing Systems*, Vol. 9, No. 4, pp.265-289, 1996.

[Z98] O. M. Zimmer and A. A. Chien, "The Impact of Inexpensive Communication on a Commercial RPC System," submitted, 1998.