Zero-Copy for CORBA — Efficient Communication for Distributed Object Middleware

Christian Kurmann, Thomas M. Stricker
Laboratory for Computer Systems
ETH - Swiss Federal Institute of Technology
CH-8092 Zürich, Switzerland
{kurmann,tomstr}@inf.ethz.ch

Abstract

Many large applications require distributed computing for the sake of better performance and software systems that facilitate the development of such applications have attracted a great deal of attention. Modeling the application as distributed objects or components promises the benefits of better abstractions and increased software reuse.

Using Distributed Object Middleware (DOM) like CORBA looks promising, but most often one cannot afford its notorious inefficiency. We address the bandwidth bottleneck by extending a highly efficient zero-copy communication architecture from the operating system through the middleware layers all the way up to the application.

In contrast to previous attempts on improving efficiency in CORBA we preserve the advantages of object oriented abstraction for the software design process and propose an efficient CORBA system that can handle bulk data transfers within the Object Request Broker (ORB). Our prototype uses a clean separation of control- and data transfers within the ORB and for the ORB-to-ORB communication and manages to get rid of all inefficient buffering for certain types while still preserving the standard Internet InterORB Protocol (IIOP). It achieves the full performance that is only available with a strict zero-copy implementation across all layers between the operating system and the application.

Keywords: Distributed Object Middleware (DOM), Zero-Copy Communication, Communication Efficiency

1 Introduction

A more and more favorable price performance ratio of commodity computing systems and the availability of high-speed networks are responsible for the popularity and the rapid development of high performance distributed computing. Distributed systems provide concurrency in execution and along with it an improvement in parallelism and performance in addition to the benefits of physical distribution, enhanced reliability and scalability. However the positive evolution of commodity hardware comes at a well known cost. The size and complexity of application software in distributed systems is increasing disproportionally. Therfore object oriented software design has attracted a great deal of attention and many middleware systems solving the most common problems have been developed. Distributed computing on the basis of distributed objects and components could eventually combine the advantages of higher performance with those of object oriented technology.

A lot of previous work has dealt exclusively with new paradigms, new functionality and establishing standard software components. Unfortunately the resulting systems were all crippled by mediocre communication performance that destroyed many benefits for high performance applications. In this paper we address the question of software efficiency and optimal performance in middleware for distributed object computing with a focus on the per-byte overheads for bulk data transfers. We chose to look into this question by the examination and the extension of a CORBA compliant software structure that relies on a well known public domain ORB. We will establish the concept that zero-copy implementation techniques are the key technology that can be used to improve the bandwidth for inter-ORB communication.

1.1 A Zero Copy Regime for Layered Software Systems

Complex layered systems, and even the most complex monolithic systems, can always be broken down into multiple levels of abstractions. The decomposition of complex monolithic systems into subsystems leads to middleware layers and eases the development by dividing the complex application into manageable parts.

In the typical layered software system, the problem of transferring data is partitioned into multiple smaller problems. This causes several characteristic problems in an efficient implementation, since the individual layers have no knowledge about the overall picture and thus may make bad decisions about how to buffer data. This is typically a contributing factor to the severely degraded and unpredictable performance of such systems [3, 7].

Figure 1 shows the control- and data path of a typical layered application with application logic, middleware for distribution, its communication system software and the operating system for resource management on each compute node or part of the system.

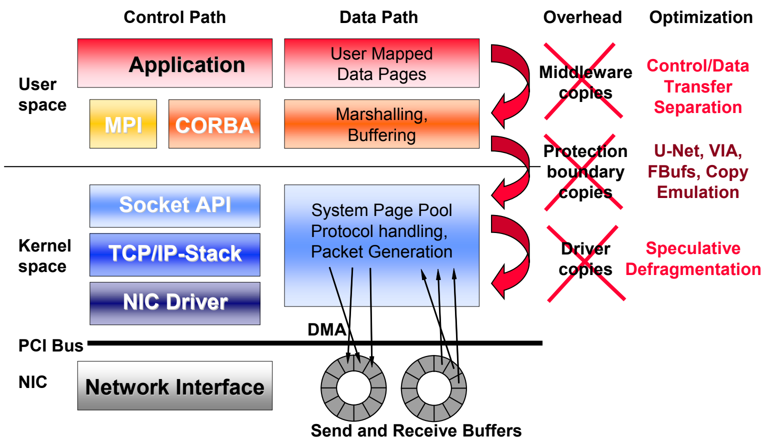


Figure 1. Control path and data path of a layered application using distributed object middleware including the typical operating system environment.

The control path in Figure 1 shows that the application communicates all its data exclusively through a middleware package like MPI [4] or CORBA [13]. The system is heavily layered, since the middleware uses a communication service of the operating system (e.g. BSD sockets) to pass the data to the underlying operating system. The OS then adds transport and network protocol headers for TCP/IP and Ethernet protocol and sends the data via the driver to the network interface hardware that takes care of the data-link and physical layer. The heavily layered control path typically introduces significant *per-packet* overheads and latency. Such overheads were recognized and removed in previous work [18]. In extension to these previous papers our study focuses on the cost of bulk transfers, since high performance distributed computing often need large amounts of data to be moved within the distributed system.

In addition to the per-packet cost some *per-byte* costs incur when data is moved within the parts of a distributed system. The typical data path in Figure 1 shows that this path is often optimized to fewer layers than the control path, but we see that in most cases several different layers and steps remain. Typically the interfaces between those steps and layers cause copies of the data. These copies introduce the main overheads for large data transfers. For the optimization of those data transfers we need to apply a strict zero-copy principle to all interfaces between the involved layers.

The copies are removed as follows. A first copy is introduced quite often by the drivers of the network interface - at least for Gigabit Ethernet using commodity network interfaces. The fragmentation of large blocks into small packets required the insertion of header and trailer information before packets can be sent and the removal thereof after packets have been received. For a simple commodity Gigabit Ethernet this copy can be removed with a probabilistic implementation technique, called speculative defragmentation [10]. In our system we reach an almost zero-copy implementation that supports the standard socket API. The most widely known copy secondly occurs at the interface between the application and the operating system. The data has to be copied from the virtual user memory space into a system page pool. This copy can be removed by a variety of rather well known techniques as stated in Figure 1. What still remains in object oriented distributed computing is a copy due to the layering of application and middleware in user space and the introduction of further protocols like the Internet InterORB Protocol (IIOP). This copy is distinct and in addition to the copy between the middleware and operating system.

This paper introduces an implementation technique to remove all copies within the ORB and to optimize the CORBA middleware for a strict zero-copy regime. Following this principle any piece of data has to be touched only once on the way from the application down to the wires of the interconnect. Unlike many previous attempts that just move copies between software layers or merge a number of copies into a single one our understanding of a zero-copy architecture really means *zero data copies* through all the involved data path layers. That means that no copy may occur in the communication system including the hardware drivers, the network and transport protocols as well as by crossing the kernel-user boundary and the involving of the socket interface and the middleware.

1.2 CORBA and High Performance Computing

Parallel programs based on message passing middleware and classical distributed systems based on CORBA often use the same distributed hardware platform - namely interconnected commodity PCs cooperating in a single application to solve a large problem. Still parallel programming systems (e.g. MPI) have evolved independently from object-based and distributed object programming systems (e.g. CORBA).

This different avenues of evolution prevent parallel programming from taking advantage of the large investment in distributed object software technologies and prevents distributed programming from reaching the efficiency of parallel

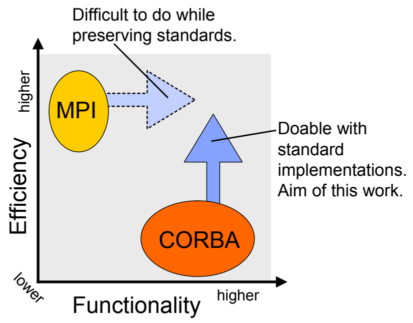

Difficult to do while preserving standards.
MPI
Doable with standard implementations. Aim of this work.
CORBA
Efficiency
higher
lower
Functionality
higher

Figure 2. CORBA versus MPI. While the efficiency of CORBA implementations can be improved, the functionality of MPI is fixed by the specification.

programs. It also makes it difficult to add parallel parts to distributed CORBA applications or distributed client/server or peer-to-peer parts in parallel MPI applications.

Several independent projects have demonstrated CORBA's usefulness for parallel programming [12, 21], generally by extending the interactions available and supporting some level of data partitioning [15, 8]. This has typically resulted in non-standard ORB extensions that are neither portable nor interoperable. But as the CORBA specification is not static, these project triggered the specification of Data Parallel CORBA [14].

On the other end MPI programs are mostly efficient when it comes to data transfers, but it is not easily possible to extend MPI towards distributed objects and to implement more functionality that eases application development without changing and enhancing the MPI specification. The tradeoffs and different approaches are depicted in Figure 2.

Based on our experience we think that we must start with the rich and well standardized functionality of CORBA and add enough efficiency to the ORB implementation to enable high bandwidth communication to the point that it can be used in high performance distributed computing.

As a communication middleware package an ORB helps the application developer to achieve flexibility and portability by taking over many common management tasks in a distributed system e.g. managing object location, parameter marshaling and object activation. CORBA is an improvement over conventional procedural RPC since it works cross platform, cross language and supports object-oriented features (such as encapsulation, interface inheritance, parameterized types, exception handling) — of course all at the cost of significant additional overheads.

To study the performance of CORBA and implement our optimizations we used one of the large and sophisticated open source projects around. MICO [16] provides a quite popular freely available implementation of the CORBA standard that is widely used. As a major milestone, MICO has been branded as CORBA compliant by the OpenGroup, thus demonstrating that open source can indeed produce industrial strength software.

2 Design Issues and Related Work

CORBA has proven its worth in low speed networks that do not demand high communication system performance. It provides a flexible solution for a heterogeneous environment, which is characteristic for many distributed systems. But at the same time the overhead incurred in providing interoperability by most middleware products adversely affects the bandwidth and latency of the system that is crucial for many parallel and real-time systems. When a conventional implementation of CORBA is applied to such systems, middleware overheads can degrade performance to such an extent that the high hardware bandwidths of a parallel system is only partially occupied or the real time system specifications are violated [18]. As a result e.g. telecommunication products are developed using proprietary middleware. Further CORBA middleware is known to be a cause of limited scalability in a number of general purpose distributed systems [1].

Although heterogeneity in distributed systems is natural, most systems are characterized by limited heterogeneity, e.g. PC clusters. Typically at least a subcluster in a system is likely to be homogeneous. We can even count on totally equal systems as a prerequisite for the best possible zero-copy operation but do allow for heterogeneity and maintain standard CORBA interoperability between the subclusters.

Other researchers have targeted CORBA applications that run on a set of similar hosts and require that large amounts of similar, non-typed or sparsely-typed data has to be communicated from one host to another. In [2] the authors give three performance optimization techniques that can essentially be characterized as bypass operations. One of the techniques prevents the data conversions between the native data types used by all system components and by the standard data format specified in CORBA.

A few previous research efforts also describe different ways of benchmarking high-performance CORBA ORBs. As part of The ACE ORB (TAO) project [18] the latency and throughput performance of a number of ORBs were measured. The figures include VisiBroker from Visigenic, Orbix from IONA, and SunSoft's IIOP reference implementation [5, 6]. The authors discuss a fully interoperable IIOP implementation and require that fine grained typed data has to be marshaled according to the rules in the specification. The performance optimization for minimal processing overheads and minimal latency of an IIOP implementation provided a first deeper insight into the problem of CORBA performance and allowed to evaluate different approaches.

We will proceed in a similar way as we try to put the system under a strict zero-copy regime for large data transfers. Still while many other systems permitted the high performance communication to be external or at least visible we will attempt to hide our optimizations and incorporate them fully into the middleware - so that the resulting API remains transparent to the application program and also the ORB-to-ORB communication remains fully CORBA compliant.

2.1 Design Issues

Detailed profiling and examination of runtime code generated for the IDL stubs and skeletons by MICO revealed that the CORBA overhead mainly stems from the following sources: data copying, request demultiplexing and memory allocation (see also [17]). Since we focus on large data transfers demultiplexing and memory allocation is not a limiting factor and the data copying is the most interesting issue to address.

There are different approaches to bypass expensive data handling. All those techniques are required but not sufficient for a strict zero-copy implementation of an ORB.

Integration of MPI in CORBA A first option to optimize bulk data handling in CORBA programs would be to integrate a full blown message passing interface library into an ORB. This route is taken by PARDIS [8]. This employs the key idea of CORBA — interoperability through meta-language interfaces — to implement application-level interaction of heterogeneous parallel components in a distributed environment.

However, PARDIS extends the CORBA object model by introducing SPMD objects representing data-parallel computations; these objects are implemented as a collaboration of computing threads capable of directly interacting with the PARDIS ORB — the entity responsible for brokering requests between clients and servers.

Legacy Code Wrapping Another approach to use a component architecture in high performance distributed computing is to simply wrap MPI-based legacy code into CORBA components. The authors of [11] describe a generator for wrapping high performance legacy codes as Java/CORBA components for use in a distributed component-based problem solving environment for a molecular dynamic simulations.

As CORBA cannot replace the MPI communication layer due to architectural and performance constraints the necessary parallelism was added to a CORBA object by running a whole MPI runtime environment inside the object to manage the intra-communication within the parallel CORBA object, and using CORBA to manage the inter-communication among objects.

Bypass of Marshaling/Demarshaling The original idea behind our optimization started with the observation that when calls are local (i.e. inside the same machine) the extra data copying that is involved by marshaling and demarshaling can be skipped. This improves the ORB latency several times. The idea can also be applied to inter-node communication in a homogeneous system. By bypassing the marshaling also the data copying within the middleware can be bypassed.

For remote communication with the same architecture on client and server, certain types, especially octets which are just 8-bit bytes, do not have to be marshaled and demarshaled at all. The negotiation of the architecture and the typeset between the client and server is specified by the GIOP (General InterORB Protocol) protocol already. We will look into this approach and an implementation later in this paper.

To achieve an ORB with zero-copy data handling we will apply all these previously known techniques and add another important implementation principle taken from high performance message passing systems in parallel computing - namely the separation of control- and data transfers.

3 The Underlying Communication Infrastructure

A low level messaging subsystem is part of all parallel or distributed systems and can transfer data from one process or thread to another even across machine boundaries. In the most general setting, such a transfer can be achieved by a wide spectrum of hardware mechanisms starting with mechanisms for shared memory interprocess communication between two time-shared tasks executing on the same processor using the same memory system to a message based communication operation between two separate processors and separate memory systems across an interconnecting network.

Although CORBA implements an RPC style synchronized client server paradigm at the higher conceptional level the predominant communication technique used inside the middleware itself is mainly the exchange of messages. By looking closer into the internal communication service one discovers that typical optimization techniques of messaging environments can be applied to eliminate buffering in ORBs as well.

In a typical message passing library there exist a variety of options to organize the transfers of control and data. In the simplest case of a data transfer, the transfer is initiated by the sender node and the original data is located in the local memory of this node. The sender then invokes a send operation to transfer a message to the destination node. This destination node invokes a receive operation to retrieve the message and to store it into the local memory at the receiver.

3.1 Control and Data Transfer Messages

The basic messaging models can be defined with a single type of messages. For the extension of these basic models to the deposit model a precise distinction between control and data messages is required [19]. All messages can be classified based on their content, their length and their purpose and are separated into two classes: control- and data messages:

Control messages are linked to synchronization The transmission or reception of such a messages does not move any data, but propagates a logical assertion between the sender and the receiver. All implicit messages generated by the lower protocol layers of a message passing protocol stack are classified as control messages. Even global synchronization primitives such as barriers are best viewed as a collection of combined control messages, regardless of whether a barrier is transmitted over a regular data communication channel or whether it is performed by special purpose hardware.

Data messages contain user data As the amount of data moved by a logical message typically exceed the hardware buffers along the communication path the proper handling of data messages involves some local memory accesses at the sender and the receiver side to retrieve or store the data.

Data messages require do be delivered from user space of one process at the originator node to the user space of another process at the destination node.

3.2 Decoupling Synchronization and Data Transfers

The separation of synchronization and data transfer is a key principle to increase communication performance in parallel and distributed systems [20]. Fully decoupled and independent synchronization generates many opportunities for optimization of both the synchronization and the data transfer traffic in a network. Improvements include simple and fast communication hardware as well as simple and well-structured system software. The biggest difference is caused by delegating buffer management to the application or the middleware. Looking at the structure of CORBA applications this means that the best option to allocate and manage the buffers is by the application or the stub and skeleton code generated through the toolkit of the ORB.

In many scenarios the buffer management is handled by the middleware and it becomes possible to optimize this buffer management within the middleware implementation. This does not effect the user application although it allows for much faster communication. Instead of suggesting the use of the magic parallelizing compilers that has global knowledge about all the communication patterns of an application we rely on some partial knowledge of the programmer about the typical communication patterns within the ORB. To avoid changes to the application interface and the synchronized client server messaging model of CORBA we introduce a decoupling of synchronization and data transfers entirely within the IIOP communication system of the ORB.

Just like for high performance message passing systems the internal decoupled messaging mechanism in the MICO ORB uses the separation of control- and data transfers for the following optimization:

Target data directly to its final destination
Synchronization is required prior to data transfers if the data is targeted directly to its final destination.

Put buffers under user control In communication systems without system buffers, all data transfers are fully under user or ORB control respectively.

Eliminate the need for buffering If synchronization and data transfers are coupled, a combined control and data message may involve buffering and result in costly storage management operations. With prior synchronization of every transfer all buffering can be omitted.

Applying these optimizations to CORBA did allow us to optimize buffering within the ORB to the extent that any performance disadvantage of buffering can be eliminated from the data path within the middleware. For large transfers this means that the data is stored directly into the destination buffers.

In the Direct Deposit messaging system [20] decoupled synchronization is used. All messages are taken directly from memory (in user space) at the sender and are automatically directed to their final destination in the memory at the receiving end. If temporary data structures or buffers are used, they are under control of the middleware, the compiler or the user and all synchronization messages are generated separately.

We will use the term direct deposit to indicate transfers that can move their data directly to the destination and can therefore be handled in a zero-copy regime with decoupled synchronization. In the next section we show how to use this implementation technique successfully in the CORBA ORB to improve the performance of large data transfers over Gigabit Ethernet drastically.

4 Implementing Zero-Copy in an Object Request Broker

To implement a zero-copy ORB according to all the design principles outlined in the previous section we modified MICO, an open source ORB, for zero-copy data transfer. As space permits we are describing the most interesting implementation issues in this project.

4.1 Data Structure

All the datatypes that can be defined in CORBA Interface Description Language (IDL) are represented by a C++-class in MICO. To internally identify these types MICO allocates a unique key to each of them. This key is represented as an integer value called Type Identifier (TID).

As we are looking into per-byte overhead we can spare a lot of overhead when large blocks of data has to be moved. As a first candidate for zero-copy operation we therefore look at the CORBA type `sequence<octet>`. An octet is an 8-bit value that undergoes no marshaling, neither by the client nor by the server. A linear sequence of these octets is called an octet stream. The semantic of streams in CORBA is quite interesting, it defines an access method that allows an item of `sequence<octet>` to be accessed directly via a pointer to a memory buffer with variable size. Therefore this data type fits the needs for direct deposit handling almost perfectly.

We use the octet streams as a first base for the implementation of an optimized direct deposit ORB. With this also all more complex types like structs with streams or arrays of streams will also be optimized as the communication of the sequence of octets is always handled with the same optimized zero-copy strategy. In more specialized applications other data types, but mostly sequences or arrays of basic types,

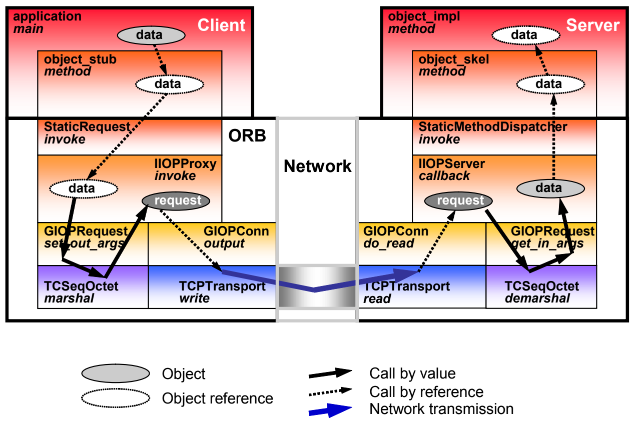


Figure 3. Data path through the MICO ORB.

might become viable candidates for zero-copy as well. For a proof of concept the type `sequence<octet>` will do as the technique would be the same for other types.

In MICO `sequence<octet>` uses a generic template `SequenceTmpl<>` which is used for all types of CORBA sequences. `SequenceTmpl<>` describes the records of a sequence by using the `vector<>`-type from the Standard Template Library (STL). This solution is very elegant and quite efficient for complex and heavily typed data structures. But for the handling of simple octet streams (which are just streams of simple bytes) such an abstraction is linked to processing that leads to a large overhead.

4.2 Data Path Through the ORB

The data path of a static CORBA method invocation within the MICO ORB is depicted schematically in Figure 3. A data transfer proceeds as follows: We assume that a client wants to communicate with a server. Therefore it allocates its data and passes it by reference through a compiler generated `object stub` and through the `StaticRequest invoke` interface to the `IIOPProxy` layer in the ORB. This is the protocol layer which implements the standardized and widely used Internet InterORB Protocol. From there the data is passed further to the `GIOPRequest` class that generates a GIOP request message by marshaling the data in the `TCSeqOctet` class. Then the `GIOPConn` class initiates a connection to the server and the request message can be sent using the `TCPTransport` implementation.

The server waits for messages to receive and uses the `TCPTransport` implementation to read the request into the buffers of the `IIOPServer`. This uses the `GIOPRequest` class which demarshals the request by using `TCSeqOctet` again. This demarshaling routine allocates the parameter data in the ORB and passes it per reference up to the `MethodDispatcher` which calls the compiler generated `object skeleton`. The skeleton maps the call to the requested server object method that implements the requested user functionality.

The major performance problem is introduced by the marshaling routine that copies the data regardless of whether marshaling is required or not (depicted by black arrows in Figure 3). The marshaling and demarshaling is handled by a virtual class that defines the virtual methods `marshal` and `demarshal`. For each of the CORBA parameter types there exist a concrete subclass that implements the required functionality by taking the parameter data and copying it to a CORBA-request buffer using an unoptimized loop and blockwise `memcpy()`. The `IIOPProxy` then generates a GIOP-request, initiates a connection, and sends the message.

On the server the data is received by the `GIOPConn` class. The `do_read` method is initiated by a callback function and runs until all the data for a GIOP-request message is received and stored into a buffer and then passed up by a further callback to the `IIOPServer`. The server then triggers demarshaling for the received GIOP-request and lets the `MethodDispatcher` call the specified object implementation and pass up the parameters. The final user method is wrapped by the server `skeleton` code which is implemented as a base class of the object implementation.

This fairly complex data path is the route that has to be optimized for zero-copy data passing within an ORB.

4.3 A Zero-Copy Datatype: The Sequence of ZC_Octet

The basis for our direct deposit optimization of large transfers is a newly introduced implementation of the CORBA-type `sequence<octet>`. To compare an optimized stream version to the standard stream version and to allow the continued use of the standard types during development we introduced a new type `ZC_Octet`, whose representation and API is isomorphic to the standard `Octet` while at the same time all corresponding methods are modified to support zero-copy direct deposit. For further data types the method of optimization would be the same.

As a first modification we had to look at the internal data representation of `SequenceTmpl<>` which is based on the C++ STL `vector<>`-type. This is not suitable for direct deposit. Therefore we had to find ways to store the data as untyped data directly in a memory buffer. We extended the definition of the `SequenceTmpl<>` class by such a buffer. Two new pointers, one to a reserved memory block, another to a page aligned area in this buffer and an integer value for the effective buffer size were added to the template class.

With this new specializations of the `SequenceTmpl<>` methods we can enable MICO to internally handle data transfers by direct deposit. The new methods are exposed to the application developer to enable the processing of zero-copy data correctly in the application. Especially useful for zero-copy programming at the application level is a `length`-method which is used for the initialization of a data block of a certain length and an new operator to access a data block.

Finally the IDL compiler has to be modified to support the new `ZC_Octet` type. Since we wanted to use both the standard and optimized octets simultaneously during the test phase we had to tell the IDL compiler to generate `ZC_Octet`

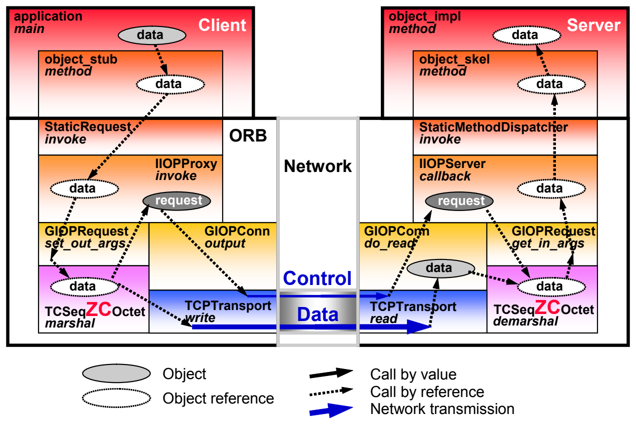


Figure 4. Data path through the optimized MICO ORB. The data copies are replaced by object references and a separation of control and data transfer is introduced.

`stubs` and `ZC_Octet skeletons`. These look the same and are used the same way as the standard sequence stubs and skeletons but use the `ZC` prefix and the appropriate type identifier `MICO_TID_ZC_OCTET`.

4.4 The Direct Deposit Sender

MICO statically instantiates methods for marshaling and demarshaling depending on the `TID` of the CORBA datatype used in the stub. It is therefore necessary to implement a new concrete class `TCSeqZCOctet` which provides the functionality for the marshaling and demarshaling.

In the case of a direct deposit (meaning a homogeneous platform is presumed) the data is never actually marshaled but just passed further on to the `TCPTransport`-layer (see Figure 4). We just need to make sure that no accidental copy is introduced. A GIOPRequest header is generated which contains the size of the data block that is needed by the receiver to correctly receive the GIOPRequest message. Here the separation of the control transfer takes place. While the GIOPRequest message header can be sent by the `IIOPProxy` over the regular networking infrastructure the data is passed to the high performance `TCPTransport` layer directly by the marshaling routine.

4.5 The Direct Deposit Receiver

The zero-copy communication at the receiver side of MICO has a totally different internal structure compared to the sender side. In MICO receiving information heavily relies on callbacks. Therefore it was not possible to integrate the data reception directly into the `TCSeqZCOctet` demarshaling routine as done in the sender. Instead of using the demarshaling routine we took the `do_read`-method in the `GIOPConn` class and separated the normal case and a direct deposit receive by introducing two distinct callbacks.

In the case of a direct deposit request the initialization supplies a memory page buffer for the GIOPRequest header. After receiving this header the receiver reads the size of the following direct deposit block and allocates an appropriately sized and aligned buffer. While receiving the parameters over a zero-copy socket interface [10] the data is directly mapped to this buffer. Afterwards a pointer is set to this buffer allowing the demarshaling routine to directly access the data and pass it further without copying.

After the CORBA-request (without direct deposit data blocks) has been received by `do_read()` the demarshaling and the method of the registered object is triggered through another callback of the `IIOPServer`. While the rest of the CORBA-request is handled normally the demarshaling routine uses the new zero-copy method for the direct deposit data. This allows to pass just pointers instead of values to the method call of the registered object. With this regime an exclusive passing-per-reference becomes feasible and copying is not needed anymore while still keeping the standard IIOP protocol and user interface.

The combination of a direct deposit sender and receiver introduces a fast data path for `sequence<ZC_Octet>` that can pass data between the CORBA application and the high performance communication system software without any copies. With this extension to the ORB middleware a true zero-copy regime becomes possible, provided the application follows the idea of zero-copy and a highly efficient zero-copy TCP/IP protocol stack is used like in our setting.

5 Performance Evaluation

As a starting point for the performance evaluation we determine the data transfer performance of an unoptimized communication between two nodes using a standard TCP/IP socket API and between two CORBA objects communicating through the MICO ORB. We use a version of the `TTCP` benchmark for the socket- and for the CORBA transfers.

Our hardware platform is a cluster of commodity PCs built from off-the-shelf 400 MHz Pentium II PCs, running Linux 2.2, connected through Gigabit Ethernet by fiber optic cables. This serves as a prove of concept as the copy bandwidth of such machines limits the communication performance dramatically when copies are involved. Our Gigabit Ethernet test bed comprises a SmartSwitch 8600 manufactured by Cabletron and GNIC-II Gigabit Ethernet interface cards manufactured by PacketEngines Corp.

For the second most interesting part of the performance test we used our highly optimized TCP/IP communication system software based on our own de-/fragmenting NIC driver using a probabilistic implementation technique [10] to handle the common case at maximal speed. This allows us to test a strict zero-copy communication mode involving the middleware and all other software in the system. Despite the high degree of optimization the communication system provides full interoperability with standard Gigabit Ethernet, standard TCP and standard IIOP communication. Any deviation from the standard is fully transparent and serves only the optimization for large data transfers.

5.1 TTCP - TCP Performance Benchmark

The data for the experiments has been produced and consumed by an extended version of the widely available TCP protocol benchmarking tool `TTCP` [22]. This tool measures the end-to-end data transfer throughput in MBit/s from a transmitter process to a remote receiver process. It can use any communication infrastructure including BSD sockets and CORBA. The following versions of `TTCP` were implemented and used as benchmarks:

Raw TCP version This is the standard TTCP program implemented in C. It uses socket calls to transfer and receive data via TCP/IP.

Zero-Copy TCP version This version replaces the default socket interface by the zero-copy sockets described in [10].

CORBA version This version substitutes all C function calls of the BSD socket interface with stubs and skeletons generated from a CORBA IDL specification. The IDL specification uses a sequence parameter for the data buffer.

We ran several series of tests that transferred several amounts of user data ranging from 4 KByte to 16 MByte in aligned 4 KByte buffers represented by a memory page. Since the zero-copy TCP-socket implementation provides its optimization for transfer sizes starting at 4 KByte pages only, the data buffers were increased by 4 KByte increments.

5.2 The Performance of Unmodified MICO

As a reference case we determine the performance of an unoptimized (or real world) data transfer in MICO. Figure 5 summarizes the TTCP benchmark results for transferring data blocks of different sizes across a Gigabit Ethernet link. It becomes evident that the CORBA-based TTCP implementation runs considerably slower than the raw TCP version programmed in C. The CORBA performance for all tests is poor

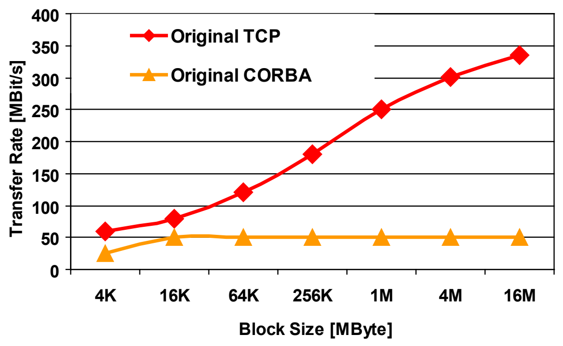


Figure 5. Bandwidths measured with TTCP for unoptimized sockets and CORBA

and reaches a saturation around 50 MBit/s out of 1 GBit/s possible in theory (sic!). The achieved bandwidths would not even use a Fast Ethernet to its limit. With the raw TCP socket an application can achieve 330 MBit/s. The limitation to one third of the theoretical speed is due to internal data copies in the TCP/IP stack. The potential for a zero-copy optimization is therefore huge if the copies in the TCP/IP stack and the copies in the CORBA middleware can be removed.

Since in this test the data block is sent as an untyped stream the CORBA presentation layer does not need to perform complex marshaling to handle byte-ordering differences between sender and receiver. Still this does not help the performance as the standard MICO CORBA implementation incurs a significant overhead.

We instrumented the ORB source code to pinpoint the sources of this overhead. The test shows that the highest cost incurs due to data copying and data inspection. Even if the transfer would qualify for a highly optimized contiguous memory-to-memory copy using MMX instructions, the portable open source project MICO uses a very general unoptimized copy loop that is able to handle all different data types correctly instead of using specialized routines that are optimized for each data types.

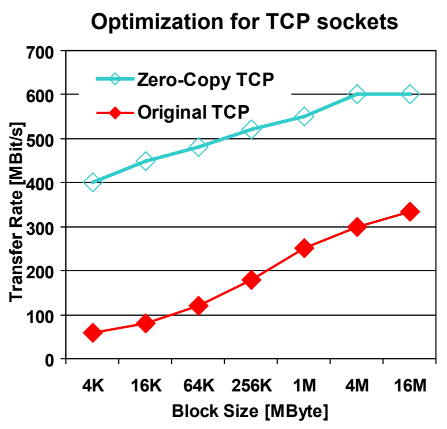
In [6] a similar analysis is done for Orbix and ORBeline where the authors found similar performance bottlenecks.

5.3 Performance Results of MICO With Zero-Copy

Figure 6 illustrates the TTCP benchmark results for the two versions of the TCP socket interface and of two different versions of the MICO ORB as well as the winning combination of zero-copy CORBA with zero-copy TCP. The performance data in the left chart shows nicely that our zero-copy TCP stack performs much better than the original copying stack. The large performance gain for small messages is achieved through a big improvement in the overhead of the `read()` and `write()` system calls. The improvement allows to achieve very good throughput figures for transfers as small as a single memory page.

The important performance improvement of the ORB implementation described in this paper is shown in the right chart. For the zero-copy version of the ORB the large overheads of CORBA are gone and the performance of the optimized zero-copy ORB nearly matches the raw TCP-socket version of TTCP. That proves that the optimized ORB handles the `ZC_Octets` correctly by passing the stream straight through the ORB without introducing any copies.

The best version of our prototype combines the performance advantages of a zero-copy TCP/IP stack with the zero-copy ORB. For large blocks this combination of ORB and protocol stack achieves 550 MBit/s throughput for large data transfers, while the application still fully complies with the CORBA model for distributed objects and components.



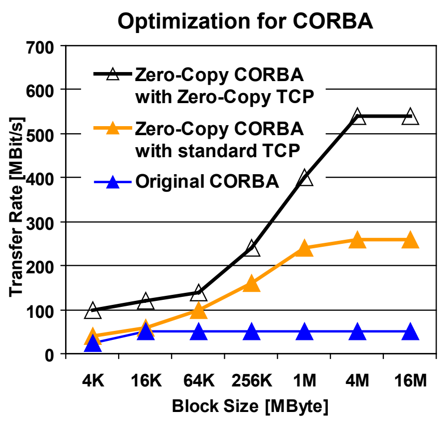


Figure 6. Bandwidths measured with TTCP for either raw TCP and for CORBA.The left chart shows the gain for the zero-copy socket interface, the right chart the gain of the zero-copy optimized ORB.

5.4 Application Performance Evaluation

Looking at the published experience reports with applications so far the use of distributed object middleware in high performance parallel and distributed computing has been fairly limited. With the optimizations introduced in this paper the middleware communication efficiency could be increased significantly making it much more valuable for high speed distributed computing applications.

To evaluate the middleware with a real-life application we introduced a service-based framework to support transparent parallelization with CORBA [9]. This allows a very short and intuitive development process resulting in zero-copy aware, parallel and distributed CORBA applications. As a technology demonstrator we implemented a real-time MPEG2-to-MPEG4 transcoder that uses the framework to parallelize an object oriented MPEG-4 encoder modeled cleanly with distributed objects. The parallel encoder objects run on a cluster of PCs equipped with Gigabit Ethernet. The video data streams that consists of a huge amount of images (or video frames) either grabbed form a HDTV frame grabber or extracted from a DVD MPEG-2 stream is distributed by CORBA requests.

We already showed the performance achievement of a factor of 10 for an optimized ORB communicating through a zero-copy operating system stack versus the original ORB. This entire performance gain is posed to our application. The resulting high performance distributed processing application provides MPEG-4 encoding in real-time for full HDTV resolution and full frame rate. Larger clusters of commodity PCs can even transcode multi-channel streams containing several parallel video streams.

6 Conclusion

Although heterogeneity in distributed systems is natural, most high performance distributed computing platforms are characterized by some very limited forms of heterogeneity. Clusters of PCs for example include machines of several generations but within a generation the machines are all alike. A general and flexible infrastructure like CORBA (Common Object Request Broker Architecture) is highly desirable but for the communication within the homogeneous parts some optimized high performance communication software is required. While a number of previous papers have addressed the processing overheads for a single remote method invocation our work improves the maximal bandwidth during the transfer of large data sets between distributed objects.

For maximal bandwidth and optimal use of the underlying communication system memory-to-memory copies along the data path must be avoided. We prove that a strict zero-copy regime can be maintained in a CORBA application using a highly optimized ORB on top of a zero-copy optimized TCP/IP protocol stack. We propose and implement such an optimized system software infrastructure that can handle untyped data in CORBA applications most efficiently and pass it between the distributed objects without extra data copies.

We propose to transfer the data with standard CORBA method calls, but optimize data handling within the ORB using a clever *Separation of Control- and Data Transfers* which enables a true zero-copy solution based on direct deposit messaging. The separation of synchronization and data transfer was a key insight that permitted better efficiency and best possible communication performance in data parallel computing. The principle is similarly effective for optimizing ORB performance in high performance distributed computing. The biggest speed improvements are caused by a clever buffer management that helps to avoid copies. For CORBA applications this means that the buffers are allocated and managed by the application or by the stub and skeleton code generated by the toolkit that comes with an ORB.

The performance results of our optimized ORB based on MICO looks very promising. The throughput results of large data transfers through our optimized ORB can match the maximal achievable performance in the transfers of raw TCP/IP-sockets. Given the limited memory performance of our testbed the good throughput figures prove conclusively that the optimized ORB handles all `ZC_Octet` sequences correctly by just passing them by reference through the ORB, without copies and without introducing undue overhead.

The zero-copy enabled ORB opens the way to an all zero-copy distributed application using a zero-copy TCP/IP protocol stack over Gigabit Ethernet. In this setting the optimized ORB still comes close to the maximal performance. For large blocks of data our all zero-copy software infrastructures achieves 550 MBit/s throughput on older 400 MHz Pentium II PCs over Gigabit Ethernet. The 550 MBit/s constitute a performance improvement of tenfold over the 50 MBit/s that are measured on the original ORB communicating over the standard TCP/IP stack. This confirms our thesis that a zero-copy regime is the most essential technique to improve the software efficiency in communication systems. For newer machines we can achieve the full communication bandwidth of Gigabit Ethernet with a CPU utilization of just 30% versus 100% with the original stack.

We tested the full functionality of our optimized communication infrastructure with a prototype of a distributed application. The CORBA based application uses a cluster of commodity PCs or a desktop grid as a highly flexible compute platform to encode MPEG-4 multimedia streams in real-time.

Despite this high performance the application code is still fully in compliance with the CORBA standard. That means that the programmer can design an application in the CORBA framework based on the concept of distributed objects and distributed components. And also for the ORB-to-ORB communication the standardized IIOP (Internet InterORB Protocol) is preserved.

References

[1] I. Abdul-Fatah and S. Majumdar. Performance Comparison of Architectures for Client-Server Interactions in CORBA. In *Proceedings of the IEEE 18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 2–11, Amsterdam, 1998.

[2] I. Ahmad and S. Majumdar. Achieving High Performance on CORBA-Based Systems with Limited Heterogeneity. In *Proc. IEEE International Symposium on Object Oriented Real-Time Computing (ISORC 2001)*, pages 350–359, Magdeburg, Germany, April 2001.

[3] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is Layering Harful? *IEEE Communications Magazine*, 6(1):20–24, Jan 1992.

[4] Message Passing Interface Forum. MPI: A message passing interface standard, Version 1.0 . http://www.mpi-forum.org, May 1994.

[5] A. S. Gokhale and D. C. Schmidt. Measuring and optimizing CORBA latency and scalability over high-speed networks. *IEEE Transactions on Computers*, 47(4):391–413, 1998.

[6] A. S. Gokhale and D. C. Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *Proceedings of the HICSS conference*, Maui, Hawaii, January 1998.

[7] V. Jacobson. Some design issues for high-speed networks. In *Networkshop '93*, page 21, Melbourne, Australia, Nov 1993.

[8] K. Keaheya and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *In Proceedings of the 6th International Symposium of High Performance Distributed Computation (HPDC)*, pages 31–39. IEEE, Aug 1997.

[9] Ch. Kurmann. *Zero Copy Strategies for Distributed CORBA Objects in Clusters of PCs*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, Dec 2002. ETH Diss No. 14950. Published by Hartung Gorre Verlag Konstanz: ISBN 3-89649-830-4 ISSN 1611-0943.

[10] Ch. Kurmann, F. Rauch, and T. Stricker. Speculative Defragmentation - A Technique to Improve the Communication Software Efficiency for Gigabit Ethernet. In *In Proceedings of the 9th International Symposium on High Performance Distributed Computating (HPDC)*, Pittsburgh PE, Aug 2000.

[11] M. Li, O. F. Rana, M. S. Shields, and D. W. Walker. A Wrapper Generator for Wrapping High Performance Legacy Codes as Java/CORBA Components. In *Proceedings of Supercomputing Conference SC2000*, Dallas, TX, Nov 2000. IEEE/ACM.

[12] OMG. RFI on Support for Aggregated Computing in CORBA. http://www.omg.org, Jan 1999. orbos/99-01-04.

[13] OMG. The Common Object Request Broker: Architecture and Specification, Version 2.4. http://www.omg.org/, Oct 2000. formal/00-10-33.

[14] OMG. Data Parallel CORBA Specification. http://www.omg.org/, 2001. ptc/2001-10-19.

[15] T. Priol and C. Ren. Cobra: A CORBA-compliant Programming Environment for High-Performance Computing. In *In Proceedings of Euro-Par 98*, pages 1114–1122, Southampton, UK, Sept 1998. LNCS, Springer Verlag.

[16] A. Puder and K. Römer. *MICO: An Open Source CORBA Implementation*. Morgan Kaufmann Publishers, 3rd edition edition, March 2000. ISBN: 1558606661.

[17] D. C. Schmidt, T. Harrison, and E. Al-Shaer. Object-oriented components for high-speed network programming. In *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems (COOTS)*, Monterey, CA, June 1995. USENIX.

[18] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[19] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proc. Intl. Conf. on Supercomputing*, pages 1–10, Barcelona, July 1995. ACM.

[20] T. M. Stricker. *Direct Deposit - When Message Passing meets Shared Memory*. PhD thesis, School of Computer Science Carnegie Mellon University Pittsburgh, May 1996. CMU-CS-00-133.

[21] John Hopkins University. Response against the Supporting Aggregate Computing RFI. http://www.omg.org, July 1999. orbos/99-07-20.

[22] USNA. TTCP: A Test of TCP and UDP Performance, Dec 1984.