Diss. ETH No. 14950

Zero Copy Strategies for Distributed CORBA Objects in Clusters of PCs

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Christian A. Kurmann
Dipl. Informatik-Ing. ETH
born December 5, 1970
citizen of Gossau (SG) and Hohenrain (LU), Switzerland

accepted on the recommendation of
Prof. Dr. Thomas M. Stricker, examiner
Prof. Dr. Burkhard Stiller, co-examiner

2002

”Communication bandwidth is like candy given to a small child
– you give the child one piece of candy
and he wants a pound.”
Hershey’s Law

Abstract

Clusters of Personal Computers (CoPs) offer the best compute performance at the lowest price. Workstations with ’Gigabit networking to the Desktop’ can enable a new game of multimedia applications that benefit from higher communication bandwidth and lower latency. In order to reach the full Gigabit/s speed on normal PCs with their typically weak memory subsystems it requires either additional hardware for protocol processing or alternatively, a highly efficient software system that circumvents data copies.

In this dissertation we successfully introduced speculation techniques into system software design and managed to implement a clean zero-copy solution entirely in software that runs with commodity network interface cards (NICs) like the ubiquitous and cheap Gigabit Ethernet adapters, using the standard TCP/IP protocol and the socket API. The implementation techniques are similar to the ones that are already widely used in the hardware design of pipelined microprocessors and should be considered to be used in software as well. Measurements and statistical studies show a huge potential for such techniques to achieve better software efficiency, that means to provide in software what the hardware promises to be able to deliver.

Distributed and parallel computing is one of the major trends in the computer industry. As systems become more distributed, they also become more complex and have to deal with new kinds of problems. To answer the growing demand in distributed software, several middleware environments have emerged during the last few years. The Component Object Request Broker Architecture (CORBA) is an example of a middleware that shows the concepts used also in many of the competing standards. These environments however typically are not implemented for being used in high speed communication settings and therefore cannot deliver the performance up to the application. Furthermore these environments often lack support for “one-to-many” communication primitives; such primitives greatly simplify the development of several types of applications that have requirements for parallel processing, high availability, fault tolerance, or collaborative work.

Since the zero-copy principle is applicable and must be rigidly used to all levels of software, we extend the design from low level drivers and protocol stack implementations to middleware packages like CORBA that ease the implementation of distributed applications. We demonstrate a study of this topic using a data and compute intensive application, a real-time distributed DVD-to-MPEG4-Transcoder, that is properly modeled by parallel objects in CORBA and still strictly adheres to the zero-copy paradigm of a highly efficient software implementation running on a Cluster of commodity PCs.

Kurzfassung

Clusters of Personal Computers (CoPs) liefern exzellente Rechenleistung zu tiefem Preis. Auch Computernetzwerke basierend auf modernen, allgemein verfügbaren Technologien bieten immer höhere Bandbreiten und kleinere Latenzzeiten. Um aber tatsächlich Bandbreiten von einem Gigabit/Sekunde in realen Anwendungen zu erreichen, benötigen solche Cluster umfangreiche Hardwareunterstützung oder alternativ hochoptimierte Softwaresysteme, welche Kopien im limitierenden Memorysystem der Maschinen vermeiden.

Wir verwenden spekulative Methoden, die auch in der Prozessoroptimierung gebräuchlich sind. Damit gelingt es uns, auch die bis anhin unumgängliche letzte Datenkopie in einem TCP/IP-Stack zu eliminieren und dadurch mit existierenden, einfachen Gigabit Ethernet Adaptern effiziente Zero-Copy-Kommunikation zu realisieren. Messungen und statistische Auswertungen zeigen, dass solche spekulativen Techniken auch in Software ihre Berechtigung haben und ein riesiges Potential zur Effizienzsteigerung ausspielen können. D.h. sie erlauben Implementationen, welche die von der Hardware versprochenen Leistungen auch in Software der Applikation zur Verfügung stellen können.

Verteilte und parallele Systeme sind mitunter einer der Haupttrends in der aktuellen Computer Industrie. Aber während die Systeme immer mehr verteilt ablaufen, werden sie gleichzeitig auch immer komplexer und kämpfen mit neuartigen Problemen. Entsprechend der grossen Nachfrage nach Unterstützung beim Verteilen von Prozessen wurden deshalb in den letzten Jahren einige sogenannte Middleware Umgebungen entwickelt. Die Component Object Request Broker Architecture (CORBA) ist ein Beispiel, welches viele ähnliche Konzepte, die auch deren Konkurrenzprodukte benutzen, vereint. Solche Umgebungen sind aber leider oft nicht für Hochgeschwindigkeitsnetze konzipiert und können daher die exzellenten Kommunikationsleistungen der Hardware nicht nutzen. Im weiteren fehlt es oft an “one-to-many”-Primitiven. Diese sind aber gerade die Voraussetzung zur Unterstützung des Entwicklungsprozesses von Applikationen, welche parallel laufende Prozesse, hohe Verfügbarkeit und Fehlertoleranz benötigen.

Da das Zero-Copy-Konzept auch auf andere Software-Schichten als das Betriebssystem anwendbar ist, zeigen wir, wie die Zero-Copy-Fähigkeit auch in Middlewareplattformen wie CORBA realisiert werden kann. Wir demonstrieren die Konzepte und Resultate dieser Thematik mit einer daten- und rechenintensiven, verteilten MPEG2-zu-MPEG4 Transkodierungs-Applikation. Diese wurde mittels CORBA und parallelen Objekten modelliert und profitiert trotzdem von der hoch optimierten Cluster-Plattform.

Acknowledgements

I am indebted to many people without whose this project would not have been possible and I would like to take the opportunity to express my sincere gratitude to all the people that generously gave me their support, time, and energy.

First and foremost, I thank my advisor, Professor Thomas M. Stricker for his patient guidance, encouragement, and confidence. Toms drive and innovative ideas have effected me deeply. It has been very rewarding and fruitful to work with him. I much appreciated the freedom I was granted to pursue my ideas and I am glad for the time that we worked together.

Special thanks go to my co-examiner Professor Burkhard Stiller for his very helpful suggestions, comments, and discussions.

Many thanks to my students Roman Roth, Michel Müller, Irina Chihaia, Rolf Laich, Matthias Ackermann, Michael Keller, Ralf Brunner and Urs Hardegger for great parts of the implementation and many stimulating discussions.

I would also like to especially acknowledge the great efforts that went into the reviewing of drafts of this thesis by my collegues Michela Taufer and Felix Rauch. The thesis has benefited greatly from their suggestions. Any omissions that remain are purely my responsibility.

Many fellow PhD students and friends have also helped and inspired me along the way, including my bike buddy Roger Weber and the CS-Gourmet&Wine-Club Jürg Bolliger, Hans Domjan, Roger Karrer, Patrik Reali and Pieter Muller. At the same time, I would like to extend my gratitude to all our colleagues on the CS-floor. All of them have brought color and good vibes to the countless Nespresso coffee breaks, lunches, dry run talks, SOLA relays, skating expeditions, assistant evenings and other social events. Many thanks to Eva Ruiz, Erwin Oertli and Viktor Schuppan for regularly organizing the “Assistentenabend” and various special events.

My dearest thanks go to my parents, Heidi and Alois Kurmann and my sister Silvia Piol-Kurmann who gave me the most valuable assets in life: love, encouragement and support.

Finally, I want to dedicate this work to my wife Nadja and my son Fabrice in appreciation of their love, understanding and encouragement. Without Nadja’s patience and support I could not have finished this thesis. Thank you for the joy of knowing you and for the wonderful person you are.

Contents

Abstract iii

Kurzfassung v

Acknowledgments vii

Contents ix

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Thesis Statement 3
- 1.3 Contributions 4
- 1.4 Roadmap 6

2 Background 9
- 2.1 Clusters of Commodity PCs 9
  - 2.1.1 Prerequisites 9
  - 2.1.2 High Performance and Supercomputing 10
  - 2.1.3 Massively Parallel and Vector Parallel Computers 10
  - 2.1.4 The Beowulf Project 11
  - 2.1.5 Commodity Off-the-Shelf Systems and Standard Software 12
  - 2.1.6 Commodity Networks for Commodity PCs 12
  - 2.1.7 Classification 13
  - 2.1.8 The Cluster Community 14
- 2.2 Cluster Networking Technologies 15
  - 2.2.1 Networking Technology Categories 15
  - 2.2.2 Cluster Networks and Supercomputer Networks 16
  - 2.2.3 Wormhole Routing Messaging Network - Myricom Myrinet 17
  - 2.2.4 Remote Mapped Memory Network - Dolphin PCI-SCI 17
  - 2.2.5 Traditional Wide Area Network - Gigabit Ethernet 18
  - 2.2.6 MPP Supercomputer Network - Cray T3D, A Reference Point 19
- 2.3 Communication with no Memory to Memory Copies 21

2.3.1 Increase in CPU, Memory and Networking Technology 21
2.3.2 Communication Input/Output 22
2.3.3 Optimizing User-Kernel Boundary Crossing 22
2.3.4 User-level Network Interface Protocols 23
2.3.5 Complex Do-It-Yourself Hardware 25
2.3.6 Copy Avoidance Techniques 26
2.3.7 Operating System Structures 27
2.3.8 User-Kernel Shared Memory 28
2.3.9 User-Kernel Page Remapping 29
2.4 Distributed Computing 30
2.4.1 Network Computing Systems 30
2.4.2 Remote Procedure Calls 30
2.4.3 Distributed Computing Systems 31
2.4.4 Message Passing 31
2.4.5 Distributed Shared Memory 32
2.5 Object-Based Computing 33
2.5.1 Concepts 34
2.5.2 Object-Oriented Frameworks 34
2.5.3 Objects in Distributed Computing 35
2.5.4 The Object Management Group and CORBA 35
2.5.5 The Object Management Architecture 36
2.5.6 The Object Request Broker 38

3 Zero Copy Implementation Strategies 41
3.1 Zero-Copy Communication 41
3.2 Communication Granularity 43
3.2.1 Communication System Comparison 44
3.2.2 Transfer Modes and their Natural Implementation 49
3.2.3 Performance Evaluation 52
3.2.4 Communication Granularity Achievable in Commodity Clusters . 59
3.3 Motivations for Zero-Copy 60
3.3.1 A Lesson about Software Inefficiency 60
3.3.2 Where Does the Efficiency Get Lost 61
3.3.3 Trends in Memory and Network Speed 64
3.4 Layered Systems 64
3.4.1 Design Strategies for Communication Networks 65
3.4.2 OSI and Internet Reference Model 66
3.4.3 The Middleware Infrastructure 68
3.5 CORBA and Parallel Computing 70
3.5.1 CORBA versus MPI 70

3.5.2 Functionality versus Efficiency 71
3.6 Extention of the Zero-Copy Paradigm 72
3.6.1 Problems with Commodity Communication Hardware 73
3.6.2 Problem with Object Oriented Middleware 73

4 Zero Copy Concepts for Commodity Hardware 75
4.1 Special Challenges of Commodity Hardware 76
4.1.1 Downside of Backward Compatibility 76
4.1.2 TCP Transmission Control Protocol 76
4.1.3 IP Internet Protocol 76
4.2 Related Work in Zero-Copy Communication with Ethernet 77
4.2.1 Zero-Copy Software Architectures 77
4.2.2 Gigabit Networking Alternatives and their Solution for Zero-Copy 80
4.2.3 Zero-Copy Extentions to Existing Technologies 83
4.3 Enabling Zero-Copy for Commodity Ethernet Hardware 84
4.3.1 Speculative Processing for Packet Defragmentation 85
4.3.2 Gigabit Ethernet and its NICs 85
4.3.3 An Implementation of a Zero-Copy TCP/IP Stack 86
4.3.4 Changes to the Linux TCP/IP Stack for Zero-Copy 86
4.3.5 Speculative Defragmentation in Hardware 87
4.3.6 Subject of Speculation 88
4.3.7 Packet Transmission and Reception 89
4.3.8 Interrupt Coalescing and Adaptive Latency Optimization 90
4.4 Performance Results 91
4.4.1 Performance Limitation in PCI based PCs 91
4.4.2 Measured Best Case Performance (Gains of Speculation) 92
4.4.3 Performance of Fallback (Penalties when Speculation Fails) . . . 93
4.4.4 Rates of Success in Real Applications 94
4.4.5 Execution Times of Application Programs 95
4.4.6 Benefit on Modern Machines 98
4.5 Improving the Success Rate for Speculation 100
4.5.1 Admission Control for Fast Transfers 100
4.5.2 Implicit versus Explicit Allocation of Fast Transfers 101
4.6 Enhanced Hardware Support for Speculative Zero-Copy 102
4.6.1 A Control Path Between Checksumming- and DMA-Logic 102
4.6.2 Multiple Descriptor Lists for Receive 102
4.6.3 Content Addressable Protocol Match Registers 103
4.7 Future Perspective 105

5 Zero Copy Concepts for Distributed Object Middleware 107
5.1 Achieving High Performance in Middleware 108
5.1.1 Performance Optimizations 108
5.1.2 Optimization Incorporation 109
5.2 Limited Heterogeneity 110
5.2.1 Standard Data Path Bypassing 111
5.2.2 Integration of MPI in CORBA 111
5.2.3 Legacy Code Wrapping 112
5.2.4 Bypass of Marshaling/Demarshaling 112
5.3 The Underlying Communication Unfrastructure 113
5.3.1 Message Passing Models 114
5.3.2 Control and Data Transfer Messages 114
5.3.3 Decoupling Synchronization and Data Transfers 115
5.3.4 Decoupling Increases Performance 116
5.3.5 Direct Deposit Messaging 117
5.4 Connecting Client and Server 117
5.4.1 The GIOP Protocol 118
5.4.2 Common Data Representation 118
5.4.3 Marshaling 119
5.4.4 CDR and Streams 119
5.4.5 Transport Requirements 120
5.4.6 GIOP Messages 121
5.4.7 The IIOP Protocol 124
5.5 MICO as a Software Platform for Zero-Copy 125
5.5.1 MICO our Software Platform 125
5.5.2 Shortcomings and State of the Art in CORBA Middleware 126
5.5.3 Data Structure Analysis 128
5.5.4 Data Path Analysis 128
5.6 Modifications of MICO for Zero-Copy 130
5.6.1 A New Datatype: Sequence of ZC_OCTET 130
5.6.2 Direct Deposit Sender 132
5.6.3 Direct Deposit Receiver 133
5.6.4 Separate High-Bandwidth Connection 133
5.7 Performance Evaluation 135
5.7.1 TTCP - TCP Performance Benchmark 135
5.7.2 CORBA Implementation of TTCP 136
5.7.3 Hardware Platform 136
5.7.4 Experimental Setup 137
5.7.5 Performance Results of MICO With Zero-Copy 138
5.7.6 Benefit of Zero-Copy Middleware 140

6 Modeling Distributed Objects for Parallel Processing 141
6.1 Parallelization with CORBA 142
6.2 Reusing CORBA Patterns for Parallel Processing 143
6.2.1 COS Event Service 143
6.2.2 COS Notification Service 147
6.2.3 Object Groups in Fault Tolerant CORBA 148
6.2.4 Load Balancing Service 149
6.2.5 Data Parallel CORBA 151
6.3 Ways to Integrate Parallelism into CORBA 153
6.3.1 Integration Approach 153
6.3.2 Interception Approach 154
6.3.3 Service Approach 154
6.3.4 Evaluation of the Different Approaches 154
6.4 Distributor Service 157
6.4.1 Design Requirements 157
6.4.2 Prerequisites 158
6.4.3 Event Communication 159
6.4.4 Integrated Load Balancing 160
6.4.5 Distributor Channels 164
6.4.6 Proxy Objects 165
6.4.7 Asynchronous, Decoupled Delivery Communication 166
6.4.8 Locking 167
6.4.9 POA Manager 167
6.5 Experiences 168

7 Zero-Copy Distributor Framework Evaluation 169
7.1 Distributor Framework 170
7.1.1 Architecture 170
7.1.2 Chunk Pool 171
7.1.3 Module Interface 171
7.1.4 Activation of Objects and Scheduling 174
7.2 Components for an MPEG Transcoder 175
7.2.1 Parallel Encoder Design 176
7.2.2 MPEG Transcoding 177
7.2.3 Partitioning of a Video Stream for Encoding 178
7.2.4 Parallelization of MPEG-4 Encoding 178
7.2.5 MPEG-4 Encoding 179
7.2.6 Collector 179
7.2.7 External Components 180
7.3 Implementation of a Distributed MPEG Transcoder 181

7.3.1 Chunk Communication 181
7.3.2 Import Modules 183
7.3.3 Process Modules 184
7.3.4 Export Modules 185
7.3.5 trans command 186
7.3.6 Example 186
7.4 Evaluation of the Results 188

8 Conclusions 191
8.1 Efficient Defragmentation with Ethernet 193
8.2 Throughput Efficient Middleware 194
8.3 Parallelization Framework for CORBA 195

A MPEG-4 Compression – An Introduction 199

B trans Manpage 203

C IDL - Load Balancing Service 207

D IDL - Distributor Service 211

Bibliography 217

Curriculum Vitae 229

1 Introduction

1.1 Motivation

The focus of novel computer architectures in Parallel- and Distributed Computing has shifted away from custom built massively parallel systems competing for world records towards some smaller and more cost effective systems built from personal computer parts. The steady and exponential growth of performance in commercial microprocessors mandates the use of these off-the-shelf components in all high performance systems and – at this time – alternative designs do not have a real chance to compete.

This trend lead to the overwhelming popularity of Clusters of PCs to run traditional supercomputer application codes. A closer look at this trend reveals a further evolution towards even more parallelism and towards a wider distribution. The better connectivity of most computers on the Internet will shortly permit a new view of things suggesting that all those personal computers and workstations form a giant global computational grid.

For the development of PC clusters the recent cost/performance gains in network technology are probably as important as the performance increase in microprocessor technology. In the field of high performance distributed computing the growth in networking enabled scientific applications that need to access large data sets over high speed networks. High data rates in the Gigabit/s range are also one of the enabling technologies for collaborative work applications like multimedia collaboration, video-on-demand and digital image retrieval. But many parallel applications for clusters are still limited by the communication bandwidth of the available communication system software. The extremely high bandwidth capabilities available by special purpose hardware challenge high performance software systems to really deliver to the user what is provided by the hardware. The ratio between sustained software bandwidth versus hardware bandwidth defines our perception of *communication software efficiency* and its optimization constitutes the most central subject of this thesis.

To achieve optimal communication software efficiency and high data rates many recent efforts have focused on designing better software architectures using the so called *Zero-Copy* principle. Well designed software is capable of moving data between application domains and network interfaces without intervention of the CPU and the memory bus intensive copy operations. This is important as the available network speeds exceeded the memory copy bandwidth of commodity systems in several epochs during the evolution of computers in the past and presumably also in the future. Unlike conventional parallel programs, the multimedia applications and tools for telecooperation mentioned above are not coded for APIs of high speed message passing libraries, but expect the standard Berkeley Socket API used in networking or they rely on standardized distributed object middleware like CORBA.

Optimal efficiency and best possible communication speed for these latter software environments is still a research issue even if the number of Fast Ethernet (100 MBit/s) and Gigabit Ethernet (1000 MBit/s) installations is rapidly growing. Ethernets became the most common means to connect computers and information appliances to the Internet. High volumes translates into low unit costs and therefore Gigabit Ethernet technology could become highly interesting for cluster computing, although the technology was clearly designed for a traditional networking world of globally interconnected networks. There are two mayor problems with this commodity technology. First Gigabit Ethernet, like all previous versions of Ethernet, has been designed for an unacknowledged, connection-less datagram delivery service. This must be fixed by a protocol stack which on the Internet is mostly the TCP/IP protocol. While IP provides the addressing and the packet fragmentation, only TCP provides a reliable, stream oriented, full duplex connection with flow control between two end points. Second, while one of the big advantages of Gigabit Ethernet is the backward compatibility with its predecessor Fast Ethernet (100 MBit/s), the downside is that its maximum transmission unit (MTU) of 1500 Bytes coming from the original Ethernet still remains smaller than the memory page size of any processor architecture.

Looking at the demands of error and congestion control mechanisms of Gigabit Ethernet, it might look completely hopeless to implement a fully compatible, fast zero-copy TCP/IP protocol stack with the existing PCI based network interface cards (NICs). A new software implementation technique is needed and with our proposed *speculative implementation* the most efficient zero-copy architecture becomes feasible, despite standard components.

Given that a zero-copy operating system environment is possible in the low level networking layers it makes no sense to introduce another copy in an upper software layer again. The last few years brought several programming environments to the marketplace that greatly reduce the complexity of developing distributed software. These environments, which we summarize under the term middleware, because they appear between application programs and operating system services, provide high-level facilities for developing distributed applications without having to deal with low-level details, such as remote communication and object location. They use object-oriented concepts to abstract the complexity of the system and promote modularity and reusability. Additionally these environments offer frameworks for the integration of heterogeneous distributed components. Examples of these middleware architectures are OMG's CORBA [112] and the CORBA Component Model (CCM), Sun Java's' RMI [163] and Java Beans or Microsoft's DCOM [19] and .NET.

Regarding feasibility of high bandwidth with these middleware implementations we also need new implementation techniques to make those software systems comply with the zero-copy regime and to enable them to communicate large amounts of data most efficiently. A better software system can be achieved by introducing *separate Control- and Data Transfers* for the remote method invocations on the middleware layer.

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. This might partly be because of the inefficiency that kills any benefit of high performance communication systems or alternatively because the above mentioned middleware packages were not created with parallel computing in mind. Looking at the conceptual issues there is no reason why a system that follows object oriented design patterns and does not contradict the goal of high performance and optimal system efficiency could not be achieved. We just have to prove that the proposed zero-copy techniques are general enough and that they do not pose undue restrictions on a clean conceptual model of parallel processing with distributed objects. A *Distributor Framework* that uses a CORBA *Distributor Service* to inherently parallelize jobs and partition data is a portable and transparent solution that allows data parallel programming with all the advantages of object and component oriented application development and deployment.

1.2 Thesis Statement

In this dissertation I claim that:

> *"An efficient communication system must be able to deliver the maximal performance of the underlying hardware to the user application.*
>
> *Efficient communication software requires a zero-copy architecture that circumvents memory-to-memory copies across all software layers in between the application and the hardware.*
>
> *A true zero-copy system architecture can be achieved even on commodity hardware for standard communication protocols."*

The validity of my thesis is proven by the construction and the evaluation of a system that masters many particular challenges in the research area of system software. A multi-layer system structure starts at the network device driver level, extends through a protocol stack, across the user/kernel boundary, uses distributed object communication middleware and ends at the top level of the application. The primary challenge to achieve zero-copy across all layers is the inherent complexity in such a highly layered system itself. The existing operating system TCP/IP stacks are not designed for high performance systems and remain therefore sub-optimal. The commodity Gigabit Ethernet NIC hardware is too simple for a true zero-copy architecture and the standard application programming interface, the Berkeley Sockets API, uses copying-semantics that are inappropriate for zero-copy.

As a second challenge the synchronous request-reply paradigm given by most object middleware makes parallelization difficult. This is one of the reasons why distributed object middleware is still rarely chosen for high performance programming of commodity clusters of PCs.

1.3 Contributions

My dissertation establishes the thesis and experimentally verifies its validity as follows:

- As a primary contribution it presents a true *zero-copy* communication architecture that includes all layers of a modern software system. It shows that optimal communication software efficiency does not need to introduce even a single memory-to-memory copy and that such an implementation can indeed be achieved transparent to the user, with cheap commodity hardware, standard programming interfaces and common distributed object middleware.

  The particular challenges related to zero-copy at the different layers of system software comprise:

  1. A **hardware driver and protocol stack layer** running with a commodity network interface card and mapping the frames containing data and protocol information to the data-link format. Like the original, this improved operating system protocol stack (i.e. TCP/IP stack) should provide a reliable streaming service upon an unreliable best effort packet delivery service of the underlying network.

  A first specific contribution of this dissertation enables efficient zero-copy communication with *commodity network technology* by eliminating the two memory copies in the standard Linux TCP/IP protocol stack while communicating over Gigabit Ethernet. This is achieved with a new approach introducing *speculation techniques* for the communication software.

  2. A **distributed object middleware layer** that facilitates the development of distributed applications for the programmer and standardizes the interface to the communication API on different operating system and different hardware platforms.

  A second contribution of the thesis is an end-to-end software system that allows to deliver data from the operating system interface through a *distributed object middleware* to any user application without using another data copy. The thesis applies the principle of *separating the control- and data transfers* of a CORBA remote method invocation. By distinguishing between control and data messages in the IIOP (Internet Inter-ORB Protocol) implementation of a CORBA request broker the zero-copy paradigm for data transfers can be maintained in this middleware layer.

3. An **application layer** in which a data intensive application must be modeled cleanly using transparent standard services for load balancing, distribution and parallel execution in a cluster environment.

At the application level my dissertation contributes a *Distributor Framework* that allows an effective development process resulting in very efficient zero-copy compliant applications.

- As a general contribution in its own right this dissertation proposes probabilistic software implementation techniques for the optimization of system software. The techniques are successfully applied to eliminate the two memory copies in a standard TCP/IP protocol stack in an area where a deterministic zero-copy implementation remained impossible given the required functionality of standard sockets and the limited capabilities of simple commodity network interfaces. With this new approach of *speculation techniques* for the communication software an "almost" zero-copy architecture becomes feasible even for simple and cheap commodity Gigabit Ethernet hardware that seemed to require copies before.

  A proof of concept for such a speculative software solution is presented and analyzed. Its core is built around a network interface card driver which initializes and uses the standard components in a much more efficient way than previous driver implementations. It optimistically speculates to process all the common cases very fast, eventually in a slightly incorrect way, and to worry about the special cases and cleanup later. For the average performance of such a solution the speculation gain, penalty and success rates have to be analyzed properly.

- Finally as an experimental study on high performance distributed applications the dissertation argues that the underlying point-to-point invocation model of object oriented middleware limits its suitability for various types of parallel and distributed applications. Therefore object oriented middleware is rarely being used in high performance parallel and distributed computing, although it offers many advantages when developing and deploying component-based distributed applications.

  With the optimizations introduced in the dissertation the middleware communication efficiency could be increased significantly making it much more valuable for high speed distributed computing applications. To further ameliorate the suitability of distributed object middleware for high performance computing this dissertation introduces a service-based approach to support transparent parallelization with

CORBA. With the proposed *Distributor Service* following object oriented design patterns parallel and distributed applications may profit from an inherent parallelization support together with all the other CORBA functionalities and advantages provided by a component based software design.

1.4 Roadmap

Chapter 2 defines the general domain of our work and the applications we are focusing on. It gives the backgrounds needed to understand the topic. An in-depth discussion of related research in the area of zero-copy architectures reveals the issues that have largely remained open so far and will consequently be explored by this dissertation. Furthermore we introduce the area of Clusters of commodity PCs, which are todays supercomputers of choice for many types of applications and we provide some short descriptions of cluster interconnects that were used as an experimental basis for this dissertation. An overview of distributed computing leads to the object oriented design models. Even as these design strategies are not yet often used in the high performance computing field, we want to show, that it is indeed possible to optimize therein a CORBA framework, so that object oriented designs for distributed applications can be implemented intuitively and still perform efficiently. An overview of the Component Object Request Broker Architecture concludes the introductory Chapter on all technologies used in the thesis.

In Chapter 3 we look into high performance communication systems and compare their natural communication modes and interfaces. The performance evaluation of this comparison shows, that for fine-grained accesses it is still needed to pack the data on the sender and unpack it on the receiver. This introduces a memory-to-memory copy. None of the currently existing communication systems was able to do the fine-grained accesses faster in hardware unlike Cray systems in the fast T3D, T3E. This confirms our thesis that zero-copy is needed especially for large coarse grain contiguous transfers. For fine-grained accesses we need packing/unpacking and therefore need copying anyway. Another reflection looking into the memory-to-memory copy problem in todays complex layered system architectures justifies the arguments that for performance reasons a *zero-copy data path through all layers* of a modern software system is essential. In the chapter the different levels involved are described in more detail. In our understanding a zero-copy architecture really means no (zero) memory-to-memory copies along the path form the network device through all the system and middleware layers up to the application. Two main problems are identified which will be addressed in the following two chapters.

Chapter 4 discusses how to achieve zero-copy with simple commodity networking hardware. Our defragmenting TCP/IP driver working over Gigabit Ethernet is based on the same *speculation techniques* that are common to improve processor performance with instruction level parallelism. Such techniques have never been considered for the implementation of standard communication stacks before. With this speculative implementation we are able to eliminate the last copy of a TCP/IP stack even on simple, existing commodity hardware. We integrated our network interface driver into the Linux TCP/IP protocol stack and added some well known page remapping and fast buffers strategies to reach an overall zero-copy communication architecture. To keep the speculation rate high we further introduced a network control architecture directly on Ethernet level. This transparent solution keeps scheduling complexity away from the application programmer and inherently tries to allocate a zero-copy blast channel on fast transfers. Based on our experience with that driver we can also suggest simple hardware improvements to the network interface to increase the speculation success rates.

Chapter 5 starts with a survey of object-based distributed systems and presents optimizations needed to a CORBA Object Request Broker to adhere to the zero-copy principle. We use the idea of *separating the control and data transfer* to achieve this. While heterogeneity in distributed systems is natural to some extent, most distributed systems are characterized by a rather limited heterogeneity. The integration of a new ZeroCopySequence type in an Open Source ORB is discussed which is used like standard Sequence types. The new type specifies a different marshaling and send routine that separates control and data transfers. The excellent performance achieved proves that the optimized ORB achieves the underlying network performance and therefore does not copy the data in memory.

Given a very efficient zero-copy environment Chapter 6 discusses some ideas on how to model distributed objects for properly parallel programming. The optimized CORBA ORB shall be used to parallelize jobs and distribute data without internal copies. We outline the requirements of such applications regarding parallelization support, present and evaluate different patterns followed by existing systems to support parallelization and load balancing in the CORBA middleware environment. A broad overview of related work leads to the observation that none of the current approaches are able to completely satisfy the requirements for optimal efficiency and they are not fully consistent with the modular, component-based architecture promoted by CORBA. We introduces the design of a *CORBA Distributor Service*. This application pattern allows transparent parallelization with decoupled clients and servers. With an inherent load balancing service the CORBA Distributor Service serves as the basic foundation to implement parallel applications on clusters of PCs with object and component based technology that still adheres to the zero-copy principle. The pattern specifies an architecture and a set of interfaces for object groups.

Chapter 7 finally introduces a *Distributor Framework* that allows a very short and intuitive development process resulting in zero-copy aware, parallel and distributed CORBA applications. By adopting this framework approach, we inherit from the major CORBA features, such as heterogeneity, portability, interoperability, modularity and reusability. The framework is further evaluated by the implementation and characterization of an MPEG2-to-MPEG4 Transcoder. This is a real-life video encoding application that uses the Distributor Framework and the CORBA Distributor Service to parallelize an object oriented MPEG Transcoder modeled as described in Chapter 6. The parallel encoder objects run on a Cluster of PCs equipped with commodity Gigabit Ethernet NICs and optimized with the speculative zero-copy TCP/IP stack as well as the zero-copy ORB.

Chapter 8 finally draws the conclusions about our work and summarizes the results of this thesis.

2 Background

The goal of this dissertation is to investigate high bandwidth communication in clusters of commodity hardware. We want to provide a framework for efficient communication with standard interfaces, even with standard middleware and object oriented design methods.

This chapter discusses the approaches that can be taken or have already been taken to achieve this goal and concludes that efficient communication with extremely high bandwidths can only been achieved when the data is not copied in-memory. The discussion of related research into zero-copy reveals the issues that have largely remained open so far and will consequently be explored by this dissertation. Further this chapter introduces the general system research area the dissertation focuses on, Clusters of commodity PCs, which are todays supercomputers of choice for many types of applications. It also surveys communication system technologies used in such clusters and in particular their network interfaces.

An attractive and fashionable programming paradigm is the object oriented design. Since this model is not yet used frequently in the high performance computing field, we want to prove, that it is indeed possible to provide a CORBA framework, to enable object oriented designs for distributed applications, which can be implemented intuitively and still perform efficiently. We therefore introduce the general concept of the Component Object Request Broker Architecture in another part of this chapter.

2.1 Clusters of Commodity PCs

2.1.1 Prerequisites

The personal computer (PC) used on every desktop has been augmented with many facilities, especially in the field of multimedia processing. Applications in this area often have a very high demand on processing power, storage and communication. The advance of PCs is based on the remarkable development of the microprocessor technology that is following Moore's law which predicts doubled performance in every 1.8 years. With the processing power the cost/performance ratio increased even more in the recent years making PCs cheaper than ever.

A second very important progress in distributed computing is caused by the large spread and the ubiquity of PCs, and the cheap networking technology resulting in today's Internet. The success of the World Wide Web and Electronic Mail lead to a coverage of the Internet which will not only allow electronic commerce but will further result in a consolidation of multimedia facilities to make the PCs an information hub in the future. Progress in technology increases the speed of networks in a local from 10 over 100 to 1000 MBit/s in the last 6 years, and commodity technologies like Ethernet reduced hardware costs dramatically.

2.1.2 High Performance and Supercomputing

As the driver for better Intra- and Internets, there are many applications that process huge amounts of transactions, such as WWW services, mail services, news services, video-streaming services or Internet searching. In addition to these applications some large-scale computation power is required for simulation and data-mining. A good example is bio-informatics, that has emerged in the past two or three years and holds promise for understanding many mechanisms of diseases and their remedies, as well as designing drugs including fundamental research on macro-molecular processes. This technology needs molecular dynamics applications and data-mining of the human genome that processes and search huge volumes of data and requires high performance computing. Another example are simulations in physics, e.g. for understanding the climate and the ocean systems to be able to predict floods, global warming and weather disasters.

The corresponding scientific simulations require huge amounts of computation power to run climate and weather models or fluid dynamics applications. To process such huge data tasks, the demand for special purpose parallel supercomputers has largely increased. In the past such systems were built with dedicated expensive hardware, especially for the inter-processor communication, today these systems are built from commodity processors.

2.1.3 Massively Parallel and Vector Parallel Computers

Cray, Inc. introduced the CRAY-1, a vector parallel computer, in 1976 and installed the first machine at the Los Alamos National Laboratory for 8.8 million dollars. It boosted the world-record speed to 160 MFlops (million floating-point operations per second). Cray and other computer vendors, such as NEC and Fujitsu, have developed so called vector parallel computers whose processing units were dedicated and could not be used as efficiently for other purposes. The design of new systems also required the development of appropriate operating systems, compilers and programming environments which resulted in expensive machines and very long development cycles. The CRAY-2 achieved 1.9 GFlops in 1985 and the CRAY C90 16 GFlops in 1991. As in the 1970s and 1980s, there were no alternatives to these high performance supercomputers and so they got very popular in the high performance computing community despite their high price tag.

After RISC-based microprocessors were introduced in the late 1980s, computer vendors such as Thinking Machine, Intel, and Cray, rethought the concept of supercomputers. The idea was to connect thousands or millions of microprocessors by a high speed network. This should enable to build high performance computing environment which should then replace vector computers for some applications.

Because neither standard high performance I/O busses nor networks were available at this time, super computer vendors started to develop dedicated high performance networking architectures, which enabled parallel supercomputers using standard RISC-based microprocessors. Such parallel computers were referred to as a Massively Parallel Computers (MPP). The first MPPs were announced in the early 1990s and could achieve about ten times the performance of a vector computer cutting down the cost of development significantly.

However, MPPs still involved the development of high performance networks and system software which kept the development cycles high in contrast to that in the PC market where the microprocessor's clock speed and performance improved much faster. The MPPs could not catch up to the latest microprocessor technology and most vendors withdrew from the market in the latter half of the 1990s.

Thanks to the performance improvements in the PC world as well as improvements in commodity computer networks it became common to build cost-effective parallel processing systems by combining of-the-shelf PCs. Such systems are called cluster systems (Clusters of PCs (CoPs), Clusters of Workstations (COWs)) or just Beowulfs.

2.1.4 The Beowulf Project

In the early 1990s Thomas Sterling and Donald Becker were working at the Center of Excellence in Space Data and Information Sciences (CESDIS) under the sponsorship of the Earth and space sciences (ESS) project. The ESS project is a research project within the High Performance Computing and Communications program. One of the goals of the ESS project is to determine the applicability of massively parallel computers to the problems faced by the Earth and space sciences community. To address problems associated with the large data sets that are often involved in ESS applications they built a cluster computer consisting of 16 DX4 processors connected by channel bonded Ethernet and called their machine Beowulf [12, 139].

The machine was an instant success and their idea of providing commodity off-the-shelf (COTS) base systems to satisfy specific computational requirements quickly spread through NASA and into the academic and research communities. The development effort for this first machine quickly grew into the Beowulf Project [11] .

2.1.5 Commodity Off-the-Shelf Systems and Standard Software

The COTS industry provides fully assembled subsystems (microprocessors, motherboards, disks and network interface cards) and the mass market competition has driven the prices down and the reliability up for these subsystems.

The development of publicly available and robust systems, in particular the Linux operating system, the GNU compilers and programming tools as well as the MPI and PVM message passing libraries, provided the scientists with all the hardware independent software they needed. With this de facto standard software suite the programmers have the guarantee that the programs they write will also run on future clusters — regardless of who makes the processors or who manufactures the networks. A natural consequence of combining the common system software with common vendor hardware is that the system software must be developed and refined only slightly ahead of the application software. In most cases bundling vendor software and hardware renders the system software to be perpetually immature. The experience with Beowulf system software contributed that rule.

Many research programs have produced years of experience working with parallel algorithms but this did not automatically lead to high performance systems an had little impact. The owners experience showed that obtaining high performance, even from vendor provided parallel platforms was hard work and required researchers to adopt a do-it-yourself attitude to systems building. This was in part responsible for the crisis in the MPP industries during the late 1990ies.

A second aspect of working with parallel platforms is an increased reliance on computational methods in many sciences and therefore an increased need for high performance computing. One could argue that the combination of these conditions (hardware, software, experience and expectation) provided an environment that makes the development of compute clusters seem like a natural evolutionary event.

2.1.6 Commodity Networks for Commodity PCs

We are constantly reminded of the performance improvements in microprocessors, but perhaps as important to the development of PC clusters is the recent cost/performance gains in network technology. The long history of Multiple Instruction/Multiple Data MPP machines (multiple autonomous processors simultaneously executing different instructions on different data) includes many academic groups and a few commercial vendors that have built multiprocessor machines based on what was then the state-of-art microprocessor, but they always required special “glue”-chips or complex interconnection schemes. For the academic community this lead to interesting research and the exploration of new ideas, but usually resulted in dedicated machines which life cycles that were too strongly correlated to the life cycle of the graduate careers of those working on them (or tenure cycles of the professors that supervised them :-)). Vendors usually made choices for special features or interconnection schemes to start out with certain characteristics of their machine or to tailor a machine to a perceived market. To exploit these enhancements required programmers to adopt a vendor specific programming model which often lead to dead ends with respect to software development. So by the late 1990ies something had to change.

The cost effectiveness and Linux support for high performance networks in PC class machines has enabled the construction of balanced systems built entirely of COTS technology which has made generic architectures and programming model practical.

In 1998, a good choice for a balanced system was 16 Dual 400 MHz Pentium II processors connected by Fast Ethernet and a Fast Ethernet switch like our first Cluster of PC (CoPs). But the exact network configuration of a balanced cluster does continue to change and will remain dependent on the size of the cluster and the relationship between processor speed and network bandwidth as well as the current market price for each of these components.

An important characteristic of clusters is this ready adaptation to the component market — processors type and speed, network technology, relative costs of components — do not change the programming model. Therefore, users of these systems can expect to enjoy some more forward compatibility than experienced before with the MPP systems.

However, until short the performance of Beowulf clusters was limited compared to that of commercial parallel supercomputers even with higher processing speed. This was especially due to the networking hardware limitations of commodity hardware. Of course there were special purpose networking technologies like Myrinet, SCI or Quadrics but these cannot been considered for cheap Beowulfs that optimize raw CPU power for the Dollar.

With the appearance of Gigabit Ethernet, Gigabit class communication has become available in commodity networks. This fact shows that the performance of commodity networks has nearly caught up with that of dedicated cluster networks.

Using Gigabit Ethernet, a high performance cluster system with application performance comparable to that of dedicated networks can be built in a LAN environment. An example is our current CoPs which consists of 16 Dual 1 GHz Pentium III nodes connected with Gigabit Ethernet by a central Gigabit Ethernet Switch.

2.1.7 Classification

Besides some experienced parallel programmer, PC clusters have been built and used by programmer with little or no parallel programming experience. In fact, clusters of PCs provide some users with limited resources (e.g. universities) with an excellent platform to teach parallel programming courses and provide cost effective computing to their computational scientists.

In the taxomony of parallel computers, clusters of commodity PCs fall somewhere between MPP (Massively Parallel Processors, like the nCube, CM5, Convex SPP, Cray T3D, Cray T3E, etc.) and NOWs (Networks of Workstations) [8]. They benefit from developments in both these classes of architecture. MPPs are typically larger and have a low latency interconnect network unlike a Beowulf cluster. Programmers of clusters are still required to worry about locality, load balancing, granularity, and communication overheads in order to obtain the best performance. Even on shared memory machines, many programmers develop their programs in a message passing style. Programs that do not require fine-grain computation and communication can usually be ported and can run effectively on Beowulf clusters. Porting an application to a NOW or the Grid [60] is usually an attempt to harvest unused cycles on an already installed base of workstations in a lab, a campus or the Internet. Programming in this environment requires algorithms that are robust against load balancing problems and extremely tolerant for large communication latency. Any program that runs on a NOW will run well on a cluster of commodity PCs..

A Beowulf class cluster computer is distinguished from a Network of Workstations by several subtle but significant characteristics. First, the nodes in the cluster are dedicated to the cluster. This helps ease load balancing problems, because the performance of individual nodes are not subject to external factors. Also, since the interconnection network is isolated from the external network, the network load is determined only by the application being run on the cluster. This eases the problems associated with unpredictable latency in NOWs. All the nodes in the cluster are within the administrative jurisdiction of the cluster. For examples, the interconnection network for the cluster is not visible from the outside world so the only authentication needed between processors is for system integrity. On a NOW, one must be concerned about network security. Finally operating system parameters can be tuned to improve performance on a cluster. For example, a workstation should be tuned to provide the best interactive feel (instantaneous responses, short buffers, etc), but in clusters the nodes can be tuned to provide better throughput for coarser-grain jobs because they are not interacting directly with users.

2.1.8 The Cluster Community

The Beowulf Project grew from the first machine and likewise the cluster community has grown from the different NASA projects. Like the Linux community, the cluster community is a loosely organized confederation of researchers and developers. Each organization has its own agenda and its own set of reasons for developing a particular component or aspect of the system. As a result, cluster computers range from clusters with a dozen nodes to clusters with several hundred nodes. Some systems have been built by computational scientists and are used in an computation setting, some have been built as test-beds for system research and others are serving as inexpensive systems to teach parallel programming.

Most architects in the community are self taught practitioners. Since everyone is doing their own thing, the notion of having a central standardization body does not make sense. But the community is held together by the willingness of its members to share ideas and discuss successes and failures in their development efforts. The future of the Beowulf alike projects will be determined collectively by the individual organizations contributing to the project and by the future of mass-market commodity PCs. As microprocessor technology continues to evolve and higher speed networks become cost effective and as more application developers move to parallel platforms, the Beowulf project will evolve further to fill its niche.

2.2 Cluster Networking Technologies

A few years ago Cray Research announced its T3D MPP line and set high standards for Gigabit/s SAN (System Area Network) interconnects of microprocessor based MPP systems sustaining 1 Gigabit/s per link in many applications. Today the communication speed is still at about a Gigabit/s, but major advances in technology managed to drastically lower costs and to bring such interconnects to the mainstream market of PCI based commodity personal computers. Many products are available today where we look at a selection of three technologies which are different in its types: The Myricom's Myrinet, Scalable Coherent Interface (SCI) and Gigabit Ethernet. All networking technologies include cabling for SAN and LAN distances and adapter cards that connect to the standard I/O bus of a high end PC. All technologies can incorporate crossbar switches to extend point to point links into a network fabric. Myrinet and Gigabit Ethernet links are strictly point to point while SCI links can be rings of multiple nodes that are possibly connected to a switch for expansion.

This section discusses some different high speed cluster interconnects used for my studies and compares them with an MPP supercomputer class network.

2.2.1 Networking Technology Categories

Two of the most promising networking technologies for interconnecting compute nodes at Gigabit speeds in a high-performance Cluster of PC's (CoPs) are Myricom's Myrinet[16], Dolphin's Scalable Coherent Interface (SCI)[41, 1]. Of course this list is by no means complete. There are a few old interconnect technologies that are not as actively marketed like e.g. the memory channel or there are other technologies that are newly announced or even shipping and we were just unable to evaluate like Quadrics [124] and Giganet [75] which resemble the SCI technique as they implement remote memory mapping technologies.

Whereas nearly all types of supercomputer backplanes or special purpose clustering networking technologies provide link level flow control mechanisms and error detection at the lowest hardware level to guarantee reliable transmission between the routing nodes traditional wide area networks (i.e. switched Ethernet) do usually not include such facilities. Any hardware flow control on their links and overflow or congestion problems result in a loss of packet in the intermediate switches and require end-to-end retransmission of lost data. Protocols capable of detecting loss and responsible for packet retransmission necessarily involve state at both ends of a data transmission. Because of this state they are said to be connection oriented.

In general a connection oriented network architecture does not fit the message passing libraries very well. Still many public implementations of the popular message passing APIs (e.g. MPI) can operate using reliable TCP streams as their means of communication, but their performance is limited by a fundamental mismatch of their model and the underlying communication structure. A conventional networking interconnect, running on standard protocols like TCP/IP over Ethernet or ATM are used under totally different conditions. For those much more flexible networks the APIs of choice are TCP streams (sockets) or RPC stub generators. Message passing libraries that operate directly on the hardware are rarely used on this class of networks.

But nevertheless these commodity networks, especially Gigabit Ethernet [136], bring "Gigabit Networking to the Desktop" which is one of the enabling technologies for collaborative work applications like multimedia collaboration, video-on-demand, digital image retrieval or scientific applications that need to access large data sets over high speed networks.

Unlike conventional parallel programs, these applications are not coded for APIs of high speed message passing libraries, but for the socket API of a TCP/IP protocol stack. The number of Fast Ethernet (100 MBit/s) and Gigabit Ethernet (1000 MBit/s) installations is rapidly growing as they become the most common means to connect workstations and information appliances to the Internet. High volumes translate into low unit costs and therefore Gigabit Ethernet technology could become highly interesting for cluster computing, although the technology itself was designed for a traditional networking world of globally interconnected computers (i.e., the Internet).

A brief survey will explain the important characteristics, properties and differences of interconnects suitable for high speed clusters.

2.2.2 Cluster Networks and Supercomputer Networks

Since the mode of operation of an SCI interface connected to a PC is so similar to the hardware of a processor node in a Cray T3D system we provide a short description of its communication technology as a reference for mechanisms and services as well as performance (see in Section 3.2). The Cray T3D is a non-commodity platform, that reached the end of its life cycle in the mean time, but is still faster (and still much more expensive) than any of its competitors today.

The main difference between the old T3D and the newer commodity interconnects for PCs is found in the indirect access to the network and the memory through an I/O bus. The "motherboard" chipset of a PC which includes the memory controller and an I/O-Bus bridge assumes the role of a main internal switching hub of the system. All I/O-operations including Gigabit networking must be performed over the PCI-bus, a standard I/O interface with lower bandwidth than the host- and memory-bus. With the advent of games and virtual reality the bandwidth requirements for graphics has outgrown the PCI bus and the PC industry reacted with a separate graphics port (AGP) for the newer PCs. Unfortunately networking seems less important to the mass market and has not yet been accommodated with a special port by the motherboard chipsets. The Cray T3D allows direct access to the memory via a highly optimized deposit/fetch engine. In addition to that engine a so called DTB annex supports direct communication from the processors caches to the network via an address translator and a few special FIFOs. The principle difference between the two commodity systems, Myrinet and SCI lays in the large amount of staging memory and the fully functional RISC processor core provided on every Myrinet interface board versus the simple buffer-oriented network interface hardware on the SCI network interfaces.

2.2.3 Wormhole Routing Messaging Network - Myricom Myrinet

The Myrinet technology is a SAN or LAN networking technology based on networking principles previously used in massive parallel processors (MPPs) [16]. Myrinet networks are built from links that carry a pair of full duplex 2 GBit/s channels that connect host and switches point-to-point. Wormhole routing with link level flow control guarantees the delivery of messages despite congestion, the checksums are just for the detection of electrical errors. The 4 to 128 port switches of Myrinet may be connected among each other by links in any topology. Myrinet packets are of arbitrary length and therefore can encapsulate any type of packet (i.e. Ethernet packets, IP packets, MPI messages) and most notably - the maximum length of the packet (MTU) is not limited. In the network link level flow control guarantees integrity of the data transfers at the expense of an increased potential of mutual blocking and deadlocks in the switches.

A Myrinet host adapter contains a LANai chip with a RISC Processor core, several DMAs and the entire network interface integrated in one VLSI chip. In addition to the LANai there are some MByte of fast SRAM on the adapter card to store a customizable Myrinet Control Program (MCP) and to act as staging memory for buffering packets.

2.2.4 Remote Mapped Memory Network - Dolphin PCI-SCI

The primary goals of the Scalable Coherent Interface (SCI) technology is to provide bus functionality with point-to-point interconnects including scalable cache coherent shared memory between physical distributed processors and memory systems (IEEE Std 1596-

1992 [1]). SCI supports a variety of topologies including rings and switched rings. The current versions of most PCI-SCI adapter card only implement a subset of the IEEE standard excluding the hardware cache coherency protocols. For full coherency there are a few expensive adapters available that replace one Pentium processor in multiprocessor motherboard and connect directly to the processor bus. The most simple full duplex connection can be established with two unidirectional links each allowing 1.6 GBit/s throughput.

We examined Dolphin's PCI-SCI adapters [41]. Those adapters currently support two modes of operation, one mode for per-word, shared memory operation (by transparently forwarding requests and responses between PCI busses) and one mode for block operation in message passing (executed by DMAs). In the first case, a load/store request to the remote memory is sent the adapter card on the PCI bus instead and translated into an SCI read/write request, sent to the remote PCI bus of the receiver and executed there as a memory operation with a potential return of data to the sender (remote reads). At the receiver a Read/Write requests to the remote memory segment is mapped to, is forwarded to the memory system through the local PCI bus.

The card consists of two main parts: the protocol engine and the link controller. The sent data with its management information is stored in eight 64Byte stream-buffers selected by the store address. This allows the hardware to gather contiguous data and combine it to a single SCI data packet. The packet is either transmitted when the stream-buffer is full, by an explicit buffer flash or an implicit timeout. With this pipelined transmission, high data rates can be achieved for contiguous data. PCI-to-SCI memory address mapping is also handled by the protocol engine with an Address Translation Table (ATT) where 32-Bit PCI addresses are converted to global 64-bit SCI addresses. Since SCI protocols are bus protocols, this mapping is seamless much like a PCI bridge and can be performed with low overhead. The most significant 16 bits of the SCI address are used to select between 64K distinct devices. This SCI address space is then mapped into user space. In addition to the logic for single word/single cache line transfers, blocks can be handled by a DMA engine that moves data transparently from local memory of a sender to the receiver. The SCI link controller chip provides flow control and guaranteed delivery of SCI packets.

2.2.5 Traditional Wide Area Network - Gigabit Ethernet

Standard Gigabit Ethernet [136] is the latest speed extension of the well known Ethernet technology. Gigabit Ethernet is successful for several reasons. The technology is simple, which translates to high reliability and low maintenance cost as well as a reasonable cost of entry (e.g. compared to ATM). As the basic technology has remained, this makes Gigabit Ethernet networks completely compatible with existing slower Ethernets. Gigabit Ethernet, like all previous versions of Ethernet, has been designed for an unacknowledged, connection-less delivery service and provides point-to-point connections supporting bi-directional communication including link-level (but no end-to-end) flow control.

Therefore it demands of error and congestion control mechanisms which are normally solved by a TCP/IP protocol stack.

But backwards compatibility also brings problems, as the original 1500 Byte maximum packet length has not been increased, which makes the interrupt rate high on such networks unless interrupt coalescing is used. Compared to the other technologies presented in this section, the latency of Ethernet is high ranging from 20 μs to several hundred μs depending on the NIC hardware, network speed and OS.

Gigabit Ethernet is layered on top of the already developed and tested physical layer of enhanced ANSI standard Fiber-Channel optical components that are well proven for connecting a network host-adapter to a central high performance packet switching backplane.

Our NICs (PacketEngines GNIC-II) use the Hamachi Ethernet interface chipset that is a typical Gigabit Ethernet controller. Besides some buffering FIFOs towards the link side, the controller chip hosts two independent descriptor-based DMA processors (TX and RX) for streaming data to and from host memory without host intervention hereby reducing the CPU load necessary for network handling. Advanced interrupt coalescing techniques reduce the number of host interrupts to a minimum and multiple packets can be handled with a single interrupt. The controller chip also detects TCP/IP protocol frames and correctly calculates the necessary checksums while forwarding the packets to the host. Some large internal FIFOs and an extended buffering capability in external SRAM chips maximize the autonomy of operation and limit the chances of packet loss.

The most appealing feature of Ethernet is its price. The cheapest Gigabit Ethernet NICs can be obtained for far less than adapters of competing technologies.

2.2.6 MPP Supercomputer Network - Cray T3D, A Reference Point

The Cray T3D node is an interesting example of an old style multiprocessor node architecture specifically designed for a distributed and parallel system. Although the original design is already retired at this time of emerging PC clusters, it still sets the standards for a good communication interface. The implementation is done in a heat producing, expensive bipolar ECL gate array technology that does not impose any compromises for cost or for standardization. There was no commercial pressure to use a PCI bus between the processor an network interface and Cray even built its own chip foundry to achieve a shorter turn around on the gate arrays used in the network interface.

The processor board comprises a 150MHz 64bit DEC Alpha EV-4 microprocessor (21064), a local memory system, a memory mapped network interface to send remote stores to the network, and a fetch/deposit engine usually names “the annex” according to the construction plans of the vendor and according to other publications. The memory of a T3D node is a simple memory system built from DRAM chips without extensive support for interleaving and pipelined accesses. Unlike DEC Alpha workstations, the node has no virtual memory and runs with a special version of the DEC Microprocessor without the functional units for paged virtual memory.

The interface between the computation agent and the main memory is centered around an 8KB primary cache and a write back queue (WBQ) which are integrated on-chip within the DEC Alpha microprocessor. An external read-ahead circuitry (RDAL) can be turned on by the programmer at load-time to improve performance of contiguous load streams; we have measured improvements of approximately 60% with read ahead. For writes, the default configuration of the cache is write-around, and support for writes consists of the write back queue provided by the microprocessor. The documentation of the Cray T3D Application Programmers Course [34] specifies the local read bandwidth at 55 MB/s for non-contiguous single word transfers, and up to 320 MB/s for contiguous reading of cache lines with read-ahead. The latency of a load from main memory is around 150 ns.

The interface between the processor and communication system on the Cray T3D consists of the annex, a memory mapped communication port, which maps some range of free address space to the physical memory of another node in the system; this node is then selected as a communication partner. The communication partner can be switched with a fixed overhead by modifying the appropriate annex entry. The significant fixed cost for switching the communication partner justifies our classification of the T3D as a highly advanced distributed memory, message passing machine. Once a store operation is issued to the communication port, the communication subsystem takes over the specified address and data, and it sends a message out to the receiver. Remote loads are handled in a similar way and can be pipelined with an external, 16 element FIFO queue. This queue requires direct coding support by the programmer or compiler and is rarely used.

At the passive end the fetch/deposit engine completes the operation as a remote load/ store on behalf of the user at the other node. These accesses happen without involvement of the processor at the receiver node (i.e., there is no requirement to generate an interrupt). This circuitry can store incoming data words directly into the user space of the processing element, since both address and data are sent over the network. The on-chip cache of the main processor can be invalidated line by line as data are stored into local memory, or it can be invalidated entirely when the program reaches a synchronization point.

Transfers from the processor to the communication system can be performed at a rate of approximately 125 MB/s, and if multiple nodes perform remote stores of contiguous blocks to a single node, these transfers can be processed at the full network speed (160 MB/s) [104]. The number of network nodes is only half the number of microprocessors (or processing nodes). If just one of the two processors is communicating at a time, the network can be accessed at up to 125 MB/s, if both processors are communicating the full speed of a network access is available and each processor obtains about 75 MB/s in bandwidth to access the network.

The interconnect topology for data transfers in the T3D is a three dimensional torus with dimension order wormhole routing. Service and IO nodes are inserted into the regular grid in at least two dimension, so that they can be reached by dimension order routing without any problems. Routing is fully deterministic and determined by a global hardware routing table loaded at boot time. There are several sets of virtual channels to permit full torus routing with a dateline and also to permit a complete separation between operating system traffic and user program traffic.

2.3 Communication with no Memory to Memory Copies

2.3.1 Increase in CPU, Memory and Networking Technology

Computer technology has advanced at phenomenal rates given its relatively recent birth, and this rate of advance appears likely to continue. This progression, however, is highly uneven, with some areas experiencing much more rapid improvement than others. A major change has occurred in the relative speeds of processors, memory, and devices. All have become faster, but memory has not kept pace with processors or devices. In modern systems, memory is often the bottleneck that limits overall performance. Furthermore, not only is the gap between memory and processor speeds growing, but the rate of this growth has increased in recent years. Reported growth rates for processors range from 40 to 100 percent per year, while memory latency performance grows by 7 percent per year (Figure 2.1).

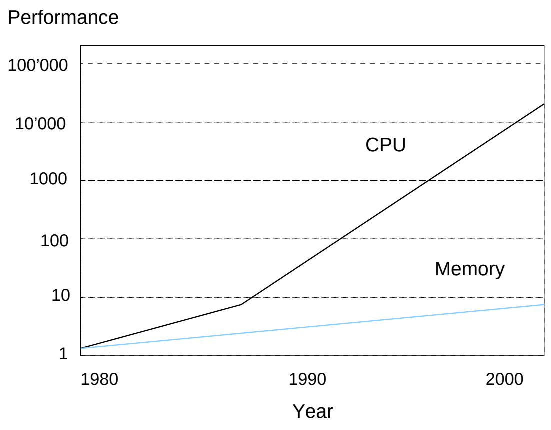


***Figure 2.1**: Memory system performance increase versus CPU improvement. (Memory Graphic Hennessy Patterson 3rd Edition) [71]*

In nearly all modern computer systems, slow memory stands between fast processors and fast, high-bandwidth devices such as RAID (Redundant Array of Inexpensive Disks) storage systems, Gigabit Ethernet or Myrinet networks and digital media devices. Although there exist operating system techniques which do not unconditionally copy data within memory during I/O (Input/Output), these all suffer either from feature compromises, such as a loss of protection, or from limited applicability, such as specialized programming interfaces.

2.3.2 Communication Input/Output

The traditional arrangement of software in a computer system divides functionality between two domains, that of the operating system (OS), or kernel, and that of user-level processes. An operating system provides basic services for the management of physical resources such as memory and processor time. It also encapsulates low-level services in a convenient, abstract representation. User-level processes provide richer functionality by making use of the resources offered by the operating system. Another issue is the security that is brought by the separation of user and kernel levels. Where the kernel code can access all data and physical devices, user processes are kept in their own private address space with access to devices over the interface provided and controlled by the kernel. One of the most fundamental services provided to user-level processes by an operating system is the abstraction of peripheral devices.

In broad terms, I/O is usually taken to mean the movement of data to and from peripheral devices. Such peripherals include disks, networks, graphic displays, and other hardware components. Among these devices, performance characteristics and modes of operation usually differ substantially. To a programmer writing user-level code, I/O has a slightly different meaning. I/O represents the movement of data between a program and either a device, or a part of the system that resembles a device. For example, a Unix socket, on which an application performs I/O operations, could be one endpoint of a communication path that spans a network. This socket represents the network device, and I/O operations on the socket cause data to enter or leave the system via a peripheral. In this dissertation we want to focus on exactly this case, that data really has to be communicated over a physical network, that means been copied from one machine to another. We won't look deeper into local inter-process communication.

2.3.3 Optimizing User-Kernel Boundary Crossing

The division of a system into user and kernel portions remains highly desirable, because of the simpler view on the system that is gained, and in most cases, because of the increased reliability gained from protection. However, user-level processes are separated from devices by an I/O system that generally depends on in-memory copying. Especially in older systems, the processing of the messages by the kernel caused multiple copies and many context switches which increased the overall end-to-end latency largely. The failure of memory speeds to match the growth of processor and device speeds has caused in-memory copying to become the principal impediment to fully exploiting the potential of modern processors and devices.

Recent efforts have been focused on designing optimized software architectures called *zero-copy*, capable of moving data between application domains and network interfaces without CPU and memory bus intensive copy operations. A variety of approaches to host interface design and supporting software have been proposed.

2.3.4 User-level Network Interface Protocols

A common reaction to the problem of User-Kernel boundary crossing is to abandon the clean separation between user processes and the kernel. Either programs formerly written at the user level are buried within the kernel. This case is often found in database server systems, where query modules are pushed down to the device driver level so that they can directly (and more quickly) examine incoming blocks of data. The drawback: Forcing user-level functionality into the kernel requires a great deal of effort. Portability and generality are lost when functions are mixed into the kernel. Furthermore, modifying a kernel is usually much harder than writing independent code.

The more common case in experimental systems is to elevate low level functions formerly hidden in the kernel to the user level. Like this the kernel boundary can be moved from the critical path of the message. User-level processes can then directly access device registers to perform I/O operations without kernel intervention. This means that parts of the protocol or the entire protocol moved to the user space from the kernel space.

The SHRIMP Project (Scalable High-performance Really Inexpensive Multi-Processor) [46] studied how tho build systems to deliver performance competitive with or better than commercial multi-computer servers. The topics are protected user-level communication, efficient message passing and shared virtual memory. Virtual Memory Mapped Communication (VMMC) [45] is a communication model providing direct data transfer between the sender's and receiver's virtual address spaces. VMMC provides protected user-space communication in a multiprogrammed environment, and was designed to minimize the software communication overhead, especially at the receiving side. VMMC tolerates network errors and guarantees in-order message delivery. Flow control, if needed, is the responsibility of higher-level software, since the sender deposits data directly in the receiver's memory.

Generic Active Messages AM [158, 156] provide the user with a model where control and data transfers are integrated. Each message specifies a remote handler that will be run upon receipt of the message. Restrictions are im posed on the operations that can be performed in the handlers. Each request is matched with a reply. Every network operation involves a round trip message exchange. Network errors are not tolerated and flow control is used to avoid dropping messages. Each sender has a number of tokens that it can use to send messages to a particular receiver. If these tokens are consumed, the sender must wait for replies before it can send more messages. Messages are delivered in FIFO order. There is no support for multiple processes per node. AM provides protection within a node, but allows arbitrary functions to be called as handlers on the remote node; therefore, the receiver needs to trust the sender. A newer, revised version of AM, AM-II, that deals with many of the limitations of the original design.

Basic Interface for Parallelism BIP [129] is a minimal library that aims at providing raw hardware performance to its users. To achieve this, it allows direct access to all system resources and provides data transfer only (no transfer of control). It is intended for single-user systems, and supports neither protection nor multiprogramming and does not support flow control of the receive buffer.

Fast Messages (FM) [121, 122] provides FIFO message delivery, and assumes a reliable network. Each message carries a pointer to a function that consumes the data at the receiver. Since messages need to be consumed, flow control is provided between the sender and the receiver to avoid buffer overflow at the receiver. An important part of FM is the streaming interface [120] that allows the user to compose a message from non-adjacent data pieces in the user address space. This provides the user with scatter/gather type operations.

One of the first examples using commodity components (like Ethernet) is U-Net [157, 160, 161]. It especially aimed at eliminating system call overhead that in the first half of the 1990s was very expensive (e.g. 5.5 us on SuperSPARC 75MHz systems). U-Net used user-level communication on commodity networks, such as Fast Ethernet and ATM. A user-level communication facility handles the network interface hardware directly without using hardware interrupts. U-Net/MM [162] also supports zero-copy communication on ATM.

Virtual Interface Architecture VIA [47, 155], is an industry standard, high-performance communication interface for system area networks (SANs). VIA provides protected user-level zero-copy data transfer, enabling low latency and high bandwidth. The communication model includes both cooperative communication (send/recv) and remote memory access (get/put).

The design of VIA was strongly influenced by the academic on low-overhead communication research (e.g. U-Net, Active Messages, Fast Messages) as well as experience with MPPs. VIA can be accelerated by relatively inexpensive VIA-aware hardware, and such hardware will more naturally support VIA than competing communication software interfaces. Examples of VIA-aware hardware include Giganet, Synfinity, and ServerNet-II. Myrinet, because of its programmable processor and other features, can also be made VIA-aware.

The VIA API consists of a library of routines called the VI Provider Library (VIPL) and a descriptor format for descriptors that are shared by the user application and VIA-aware hardware. VIA communication routines are normally implemented entirely at user level, while other functionality (e.g. connection management and memory registration) are implemented in the operating system kernel.

Since VIA was only intended to be used for communication across the physical servers of a cluster (in other words across high-bandwidth links with very high reliability), the specification can eliminate much of the standard network protocol code that deals with special cases. Also, because of the well-defined environment of operation, the message exchange protocol was defined to avoid kernel mode interaction and allow for access to the NIC from user mode. Finally, because of the direct access to the NIC, unnecessary copying of the data into kernel buffers was also eliminated since the user is able to directly transfer data from user-space to the NIC. In addition to the standard send/receive operations that are typically available in a networking library, the VIA provides Remote Direct Memory Access operations where the initiator of the operation specifies both the source and destination of a data transfer, resulting in zero-copy data transfers with minimum involvement of the CPUs.

PM [151] the underlying low level messaging library of the SCore Cluster System Software developed by the Real World Computing Partnership RWCP in Japan (now PC Cluster Consortium) uses a daemon to multiplex communications for multiple processes over each network interface. The system was designed to support gang scheduling, so most of the protection issues are avoided by having one process access the network interface at each time. PM provides in-order message delivery and flow control between the sender and the receiver, but does not tolerate network errors. It supports two kinds of usage: a traditional send/receive model, and a remote write model. System buffers can be accessed directly, avoiding the overhead of an additional copy to/from application buffers.

All of these techniques carry penalties. Elevating low level services out of the kernel also leads to a loss of generality. For example, a device whose registers are mapped by one process cannot be simultaneously accessed in the same manner by any other process. In addition, safety and reliability are more difficult to obtain when low level operations are not kept under wraps inside the kernel. While tearing down the user-kernel boundary can provide immediate performance gains, the long-term result is usually a loss. Keeping user and kernel services separate is the best long-term strategy.

2.3.5 Complex Do-It-Yourself Hardware

Another common technique for improving I/O performance is to substantially increase the sophistication, and computing power, of peripheral devices. In the traditional architecture, the NIC would simply take the data from the host and put it on the interconnect. However, modern NICs have programmable processors and huge amounts of memory placed in the peripheral or in the hardware interface by which the peripheral is connected to the I/O bus [16, 80]. This second computer makes them capable of sharing some of the message processing work with the host, thus reducing the load on the main processor and memory. This technique yields improvements that apply only to a particular device (the one with the extra processor). These improvements usually come with a substantial penalty in design complexity, as tasks formerly performed entirely in the kernel must now be divided among two computational domains.

A different approach to copy-free I/O by peripherals takes advantage of the DMA (Direct Memory Access) hardware present in most peripheral devices. This hardware allows the device to copy data to and from system memory without direct supervision by the CPU. With careful planning, it is sometimes possible to perform these DMA transfers directly to or from the memory of a user-level process, without an in-memory copy through the kernel.

It is feasible arranging for DMA from a process to a device, although blocking the process during the transfer can reduce concurrency. The more difficult problem is the other way round, arranging for DMA from a device to a process. Network interfaces produce data unpredictably and packets may arrive in any order, for any of several receivers. Only if the interface has sufficient processing power to inspect packets and determine their destinations a DMA transfer can reliably be performed to the correct target buffer.

The technique known as kernel streaming eliminates in-memory copies during I/O by removing the user-level process from the I/O path. Data enters the kernel from one device, say a hard drive, then leaves for another device, e.g. the networking interface, without ever being copied to or from a user-level process. This technique is interesting for file- or web-servers and is implemented in Linux with the sendfile system call. In the Linux community this technique is often addressed as zero-copy streaming, even if there still exists a copy for defragmenting the file system cache pages to Ethernet frames in the TCP/IP stack. As the last copy is always the most expensive one (the data of further copies lies often in the cache), we could not measure much improvement with the sendfile implementation. In addition to eliminating all copies (in theory), further savings may result from fewer context switches and system calls. Unfortunately, few devices are suited to talking directly to each other without some amount of intermediate processing. One system which supports kernel streaming is Splice [49, 50].

Finally, there is an I/O technique even faster than kernel streaming. Bus architectures like the PCI bus enable devices to communicate directly, without passing data through the CPU or even the memory system. This form of communication is called device streaming. In practice, it is extremely uncommon for any two devices, even identical ones, to be able to communicate directly, because a great deal of processing normally provided by the kernel must be incorporated directly into the devices. If each device is enhanced with a local processor and some form of operating system, the problem becomes essentially one of multiple systems communicating over a local area network, the system I/O bus. An example of such a system is the OPIOM I/O system [63] where a Myrinet adapter directly communicates with a SCSI adapter.

2.3.6 Copy Avoidance Techniques

To preserve protection, multiuser systems usually do not allow applications to access Input/Output devices directly: Applications can perform I/O only indirectly, by explicit or implicit requests to an authorized kernel- or user-level server or pager and ultimately to an authorized driver. However, requests and their respective replies may involve significant data and control passing overheads, such as copying and context switching.

As discussed in the last section a long line of research has aimed at alleviating I/O data and control passing overhead, often proposing:

- Changing the *semantics* of data passing between applications and the operating system, so as to avoid data copying; or
- Changing the *structure* of the operating system, so that data passing and control passing between applications and operating system can be reduced or eliminated.

This section discusses specific techniques that have been proposed or implemented to avoid in-memory copying during communication preserving the data passing semantics. And still the system structure can give end-to-end improvements competitive with those of data and control passing optimizations that change semantics or structure.

Each of these techniques eliminates copies at one or more points in the system. Since copying data usually places the data in the cache, where subsequent copies are much faster, the largest performance gain can only be achieved if every single copy is eliminated, rather than if only some copies are eliminated.

2.3.7 Operating System Structures

Peripheral drivers in an operating system have to access device controller registers and other data structures that generally should not be accessible by applications. Therefore, the protection domains of drivers and applications usually must be different.

The assignment of separate protection domains for applications and drivers depends on the structure of the operating system. Such structure establishes how system implementation is decomposed into modules, how protection domains are implemented, and how protection domains are assigned to system modules and applications so as to preserve system protection and integrity.

To implement protection domains, most operating systems rely on two hardware-supported processor features: virtual memory (VM) management and privilege modes. VM hardware treats memory addresses issued by processes as virtual and automatically translates such addresses into physical ones. Physical addresses are used to access physical memory. In the most common scheme, physical memory is split into fixed-size blocks (e.g. 4 KByte for an x86, 8 KByte for an Alpha system), called pages. To translate virtual addresses into physical ones, VM hardware consults the current page table . If a process issues a virtual address for which no valid translation exists in the page table, control of the processor is automatically transferred to the system's VM fault handler by issuing a page fault trap. In systems such as Mach and those derived from BSD Unix or Linux, allocated pages belong to a memory object. Each memory object is backed by a pager. On a page fault, the handler allocates a physical page and invokes the object's pager to retrieve the contents of the virtual page into the physical page. When the pager returns, the VM fault handler maps the physical page to the faulted process. Therefore it modifies the page table so that the faulted virtual address translates into the physical address of the page. The handler then makes the faulted process again runnable.

Because the number of pages in physical memory is limited, each page allocation necessitates, in general, a counterbalancing page deallocation. A kernel-level process scans and deallocates currently allocated pages when the number of free pages in the system is low. If the daemon selects for deallocation a page that was modified after being last retrieved from its pager, the daemon invokes the pager to save the page's contents. When the pager returns, the daemon unmaps the page and places it in a list of free pages. Pagers usually save and retrieve page contents to and from storage devices, but may also do so remotely, over a network. In Mach and related systems, applications can supply their own user-level pagers when allocating a region, that is, memory spanning a given range of virtual addresses.

The correspondence between virtual addresses and pages in physical memory and backing storage devices is called an address space. The instruction to switch the current page table is usually privileged, that is, can be executed only in the processor's kernel mode. By running each application in its own address space, in user mode, systems can prevent applications from gaining direct access to each other's or the system's data (including, for example, device controller registers). Code running in kernel or user mode is also called kernel-level or user-level code, respectively. To switch into kernel mode, applications typically have to execute a special instruction, the system call, which jumps to a well defined address. The system installs its own code at such address, and sets up application address spaces so that applications cannot otherwise access (or corrupt) the memory occupied by system code. Most processors can address a wider range of virtual addresses in kernel mode than in user mode. The system address space, therefore, can be implemented as a complement to every application's address space, and no address space switching is necessary to cross the kernel/user protection boundary.

2.3.8 User-Kernel Shared Memory

The use of virtual memory generally makes it possible for two or more processes to share memory in a portion of their address spaces, while remaining otherwise isolated. This technique allows fine-grained communication, which may be asynchronous, without necessarily compromising the protection provided by independent virtual address spaces.

With the same technique it is also possible for a process to share memory with the kernel. Since the kernel may have full access to all memory, this sharing may be more a matter of definition than mechanism. By sharing a pool of memory where data is stored during I/O, both the user-level process and the kernel can access the data without in-memory copies.

Although the concept of shared memory I/O is simple, the mechanism is complicated in practice. The user-level process and the kernel must coordinate their access to the data, because there is now only one copy in memory. When new data is placed in the shared pool by the kernel, the process must learn where the data arrived. A process may have no way to take data that arrives in the pool of one device and transfer it to the pool of a different device, other than copying the data in memory. Finally, because the user-level process retains full access to the data at all times, and can modify it asynchronously, safety and security may be compromised.

2.3.9 User-Kernel Page Remapping

Another approach to copy-free I/O is to use the virtual memory system to transfer memory dynamically between the kernel and a user-level process, rather than sharing a static pool of memory. Because the virtual memory system implements protection in units of pages of memory, this technique is called page transfer, or virtual page remapping. If a page transfer system revokes access by user-level processes when a page is in use by the kernel, security problems such as those described for a shared memory system can be avoided. A page transfer system need not be bound to a per-device pool of memory, so inter-device I/O without in/memory copying may be easier. Page transfer mechanisms can still be complicated for reasons similar to those which complicate a shared memory approach. In particular, a user-level process may experience rapid modifications of its virtual address space as pages appear and vanish.

Container Shipping [7], DASH [6], FBufs [43, 44], and the mmap facility [94] all employ some form of page transfer or sharing to eliminate copies.

DASH did include a data transfer mechanism for I/O that used virtual page remapping to avoid in-memory copies. But the key feature of DASH was high-performance inter-process communication via page remapping.

Fbufs uses virtual memory manipulations to move data between the kernel and a user-level process. Offset and length information are used to locate data within page-based buffers. The Fbufs mechanism requires that all data be read-only at every point after it enters memory. This key constraint makes it impossible to perform small, fine-grained modifications on a data stream without copying the entire stream. While Fbufs can speed up I/O that does not modify data, the readonly constraint greatly reduces the generality of the solution. Fbufs also achieves great speed gains by not enforcing the protection between a user-level process and the kernel.

Container Shipping offers universal I/O facility, usable with any device and unconstrained by the device type and selective complete access controlled by the user-level process without read-only constraints, indirect access functions, or copy-on-write penalties. Container Shipping specifies only a transfer mechanism, so it can be integrated with an existing I/O subsystem rather than replacing it. Memory is never shared, so protection is never compromised.

All these IPC facilities with copy avoidance provide interfaces with non-copy semantics and therefore are incompatible with the many applications written according to that semantics. Emulated Copy [22] preserves copy semantics and can therefore be used to optimize I/O interfaces of systems such as Unix, which also have copy semantics. Emulated Copy uses input alignment for input by page swapping even when the client buffer is not page-aligned. Another optimization is transient output copy-on-write, which allows to output data in-place and with strong integrity guarantees. A detailed overview of data passing semantics is given in [21].

2.4 Distributed Computing

Computing systems have evolved from centralized architectures to distributed systems. Distributed systems have then further evolved from simple client/server applications running on Local Area Networks (LANs) to complex systems involving a huge number of machines across Wide Area Networks (WANs). This section presents a brief history of distributed computing [14].

2.4.1 Network Computing Systems

A networked application is a computer application that consists of several decoupled components communicating by exchanging messages. The development of client/server networked architectures peaked during the 1980s, when it became possible to put the power of a mainframe on a desktop computer. The concerns of network computing are generally described in terms of the Open Systems Interconnection (OSI) layering. In this descriptive structure, several software layers abstract the details of the physical network, packet and message transmission, routing, data representation, addressing, and session management. Each layer is built using the services provided by the underlying layers. The TCP/IP protocol suite is a typical example of a network architecture that is closely matched with the OSI model. A property inherent to the OSI model is that communication is limited to point-to-point data transmission.

2.4.2 Remote Procedure Calls

The next big evolution of distributed computing occurred with the introduction of Remote Procedure Calls (RPCs) [15]. RPCs allow client programs to transparently issue calls to procedures defined by remote server programs. The complexity of making connections and marshaling data in and out of messages is completely hidden from the application by stubs that mimic the interface of the procedure calls. Network operating systems have been hugely successful over the last 15 years, and RPC mechanisms have been extensively used in these operating systems for distributed services such as network file systems, name services, and synchronized clock services.

2.4.3 Distributed Computing Systems

In the late 1980s, with the availability of powerful desktop computers that can be interconnected through very fast networks, centralized multi-processor parallel architectures have been progressively replaced by distributed system architectures. The term distributed computing, in contrast with network computing, designates a set of tightly coupled programs executing on one or more computers and coordinating their actions. These programs know about one another and cooperate to perform a task that none could carry out in isolation. Such systems allow the sharing of information and resources, and may be composed of small, cost-effective computers that combine their processing power.

A typical example of a distributed computing system is the Parallel Virtual Machine (PVM) [147], which is a software package that permits a heterogeneous collection of computers hooked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved at low cost by temporarily using the combined power and memory of many computers.

2.4.4 Message Passing

Message passing has been widely adopted as the communication paradigm in the programming of distribute memory parallel systems. The Message Passing Interface MPI [57] is a standardized messaging API used for programming a certain class of parallel machines especially those with distributed memory. Such runtimes allow the sending and receiving of messages through explicit send and receive operations with various semantics (blocking or non-blocking). Messages are usually associated with a type to allow a selection at the receiving side.

Although in the past there were many variations, the basic concept of processes communicating through messages is well understood. While substantial progress has been made in casting significant applications in this paradigm, first each vendor has implemented its own variant. After several systems have demonstrated that a message passing system can be efficiently and portably implemented, in 1992 the time has been come to try to standardize both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

The design of MPI has largely been influenced by the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry.

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [42]. MPI1 embodied the main features as being necessary in a message passing standard mainly focusing on point-to-point communications. The Draft MPI standard was then presented at the Supercomputing 93 conference in November 1993 by the MPI Forum, membership of which has been open to all members of the high performance computing community. It included collective communication routines and thread safeness.

The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are build upon lower level message passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard, such as MPI, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

The goal of the Message Passing Interface was to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing. The MPI standard therefore defines an abstract library interface for message passing and bindings of this interface to major programming languages. Coming from the high-performance computing community, MPI focused on Fortran and C bindings and was enhanced with a binding to C++ later.

One major goal of the specification was to allow efficient, high-performance implementations of the interface, which make direct use of inter-node communication mechanisms in a parallel computer. That means communication offloading to communication co-processors can be implemented where available. MPI by design allows efficient communication by avoiding memory-to-memory copying and communication overlapping of computation. To ease complexity the standard assumes a reliable communication interface beneath the library that means that the user needs not to cope with communication failures. Such failures have to be dealt with by the underlying communication subsystem.

The interface works equally well in a shared-memory architecture, and on distributed memory systems (in particular on the local area network). Therefore, algorithms written for the MPI are scalable across a wide variety of computing platforms. High-performance computing platforms today often come with customized MPI implementations. In addition, a number of free implementations are available. These implementations typically allow distribution over the local network, thus supporting cluster of PCs solutions.

2.4.5 Distributed Shared Memory

Distributed shared memory systems [95] like TreadMarks [87, 5] or Cashmere-2L [140] are seen as an alternative for the programming of distributed and parallel systems. They give the illusion of a single address space in a computational infrastructure in which each node has its own local physical memory.

The OpenMP application program interface is an emerging standard for parallel programming on shared-memory multiprocessors. It defines a set of program directives and a library for run-time support that augment standard C, C++ and Fortran [73, 152, 59]. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

In contrast to MPI, OpenMP facilitates an incremental approach to the parallelization of sequential programs: The programmer can add a parallelization directive to one loop or subroutine of the program at a time. Unlike POSIX threads [23], OpenMP specifically addresses the needs of scientific programming, such as support for Fortran and data parallelism.

2.5 Object-Based Computing

While distributed computing is appealing, since it allows us to decompose and extend applications in a very flexible and powerful way, it is harder to manage because we have to address issues such as independent failures, unreliable communication, and insecure communication.

Booch [17] presents the inherent complexity of software as deriving from four elements:

- the complexity of the problem domain
- the difficulty of managing the developmental process
- the flexibility possible through software
- the problems of characterizing the behavior of discrete systems.

A common goal to most analysis and design methods is to hide this inherent complexity, and to give the illusion of simplicity. Dealing with complexity may be addressed through the use of decomposition, abstraction, and hierarchy. Object-oriented analysis and design is a method based on object-oriented decomposition where the complex problem is viewed as a meaningful collection of small objects that collaborate to achieve some higher level behavior.

Object-oriented programming today is the most widely used programming model today. The work has led to a fundamental change in how software systems are designed and programmed, resulting in reusable, reliable, scalable applications that have streamlined the process of writing software code and facilitated software programming. Current object-oriented programming languages include C++ and Java, both widely used in programming a wide range of applications from large-scale distributed systems to personal applications.

2.5.1 Concepts

The major goal of Object-Oriented Programming (OOP) [159] is to provide a better programming model for representing the world. The basis of the object model is the concept of objects, which are entities modeled on the real world, that have specific properties and exhibit specific behavior. An object is an instance of a class. Objects provide the means for combining behavior and state (i.e., program and data) into a single entity. The four major concepts underlying OOP are abstraction, encapsulation, inheritance, and polymorphism.

- **Abstraction** Abstraction is a key concept of object-oriented design. It is one of the fundamental ways to cope with complexity. An abstraction focuses on the outside view of an object and offers a simple and concise representation of a more complicated idea. The aim of abstraction is to hide complexity: complexity does not disappear, but does move to a more appropriate point in the architecture.

- **Encapsulation** Encapsulation hides the implementation details of an object from the services it can provide, by separating the contractual interface of an abstraction and its implementation. Other objects can request services by sending messages to the object that provides the service. By decoupling the interface and the implementation, encapsulation enables us to modify an object's implementation without affecting its clients.

- **Inheritance** Inheritance enables us to pass along the capabilities and behavior of one class of objects to another. Inheritance defines a relationship between a child class that inherits from a parent class. The child class can be specialized by modifying inherited methods, or adding new ones. The parent is not affected by this modification. Inheritance enables us to reuse the properties of existing objects. Some systems support multiple inheritance, in which a child class may inherit from several parent classes.

- **Polymorphism** Polymorphism is the ability to substitute objects with matching interfaces for one another at runtime. With polymorphism, objects that have common descriptions can respond to the same request with different actions, depending on their type. Polymorphism makes it possible to design applications that are easily extensible.

2.5.2 Object-Oriented Frameworks

A framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate [62]. This design is a solution for a family of problems. It describes how the software is decomposed into a set of interacting objects which have a given responsibility.

Frameworks are generally classified according to two different categories: white-box and black-box frameworks. White-box frameworks exhibit the internal structure of their class to the user. The application specific behavior is usually defined by adding methods to subclasses of the framework classes. Each method added to a subclass must abide by the internal conventions of its super-classes. Black-box frameworks are composed of components that provide the application-specific behavior. These components interact together using method invocations, so the user needs to understand only the external interface of the components.

Black-box frameworks are easier to use and learn than white-box frameworks, because the user is not required to have knowledge about internal details of the classes he uses. Black-box frameworks make it easy to change behavior at runtime, by replacing a component by another component with the same protocol. On the other hand, white-box frameworks are more flexible since they permit the definition of other behaviors than that supplied by the framework, and the number of possible combinations of components is not predetermined by the architecture of the framework. In this thesis, we describe an architecture that defines a black-box style framework.

2.5.3 Objects in Distributed Computing

Distributed object middleware (DOM) such as CORBA combines to varying degrees the functionality of all all the mentioned middleware with object-oriented concepts to extend the benefits of object-oriented software engineering to distributed computing.

Several environments provide object-oriented abstractions and frameworks for distributed computing. Some of them mainly define wrappers on top of operating system services, such as the Adaptive Communication Environment (ACE). Other infrastructures provide more elaborate facilities for distributed protocol composition. Modern object-oriented programming languages include intrinsic support for distributed objects, such as Java Remote Method Invocation (RMI) mechanisms.

2.5.4 The Object Management Group and CORBA

The OMG was established in 1989 to create standards for distributed object computing. One of its aim was to ”move software development closer to the age of components”. Its standards were intended to allow interoperability of objects, components, and applications in a heterogeneous networked environment. Early efforts resulted in the creation of the Object Management Architecture (OMA) specification. The OMA reference model as it now stands is shown in Figure 2.2.

2.5.5 The Object Management Architecture

The OMA is a cornerstone of the OMAs standards [106]. It provides a definition of object concepts and terminology, but leaves actual implementation decisions to vendors. This allows a variety of approaches to be taken to build a product. The OMA describes five components, as follows:

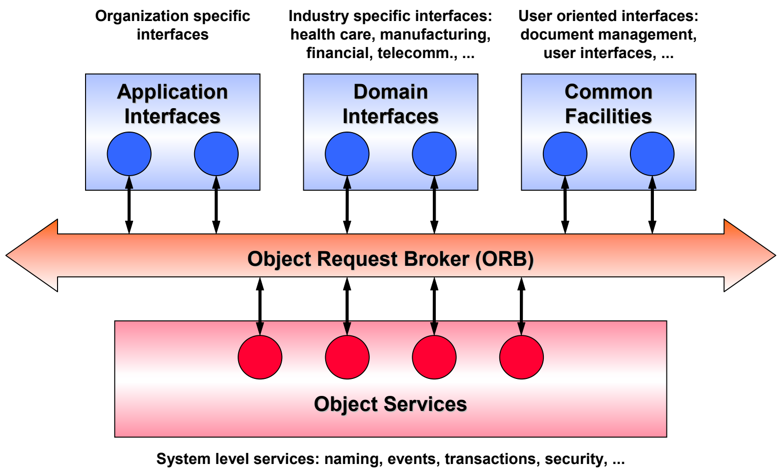

Organization specific interfaces
Industry specific interfaces: health care, manufacturing, financial, telecomm., ...
User oriented interfaces: document management, user interfaces, ...
Application Interfaces
Domain Interfaces
Common Facilities
Object Request Broker (ORB)
Object Services
System level services: naming, events, transactions, security, ...

***Figure 2.2**: The OMA's Object Management Architecture.*

- **The Object Request Broker**: The object request broker (ORB) is the core component of the OMA. It transparently allows requests from the other four components to be transmitted, using standardized interfaces. The ORB can be viewed as an "object bus", through which heterogeneous objects can interoperate. Interfaces for CORBA objects are defined by the interface definition language (IDL), a programming language-independent specification. The IDL is used to create the visible interface between a client and a server.

- **Object Services**: The object services are the set of general purpose system-level components that are fundamental in order to build robust and useful distributed object systems. A CORBA service is basically a set of CORBA objects with their corresponding IDL interfaces, that can be invoked through the ORB. Services are not related to any specific application but are basic building blocks, usually provided by CORBA environments, supporting basic functionalities useful for most applications. Several services have been designed and adopted as standards by the OMG.

- **Application interfaces**: Application Interfaces are the interfaces that are specific to an organization or application and are developed during system construction. An interface is a description of a set of possible operations that a client may request from an object. An object satisfies an interface if the object can be specified as the target in each potential request described by the interface.

- **Common facilities**: The Common Facilities provide end-user-oriented capabilities useful across many application domains, that can be configured to the specific requirements of a particular application. These are facilities that sit close to the user, such as printing, document management, and electronic mail facilities.

- **Domain interfaces**: Domain Interfaces represent vertical areas that provide functionality of direct interest to end-users in specific application domains, such as finance or health care.

The OMA specifies a classical abstract object model. According to this model, a client submits a request to an object. A client is an entity that requires a service to be performed outside of its internal definition. An object, also called an object implementation or server, performs one or more services for a client. The request consists of an operation - also called a method - a target object, and zero or more parameters. The interface is the description of operations supported by the object with the associated request format. IDL is used to define an interface between clients and servers.

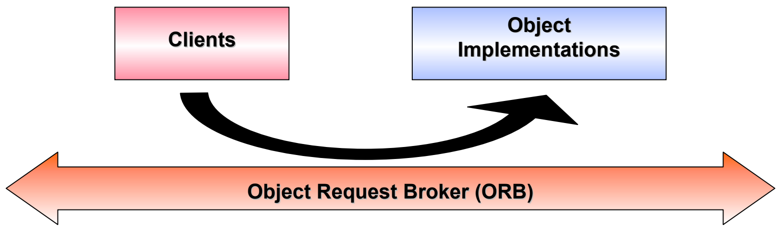


Figure 2.3: *The ORB handles requests to objects.*

CORBA is the standard that specifies a concrete object model based on the OMAs abstract model. Its basic function is to handle requests between clients and object implementations, as shown in Figure 2.3. The ORB handles these requests, either within a single computer or across a network in a transparent fashion.

CORBA has several mechanisms to handle these requests. The client can use the following: IDL stub. An interface that is completely defined in the IDL an tied to a specific target object. Dynamic invocation. A general interface that is independent of the target object interface

The client also can talk to the ORB, but this is used only rarely., The client makes a request by having an object reference, a pointer or address on a network, for an object with a desired service. The client either sends the request through the static IDL subs or generates the request and sends it through the dynamic invocation.

2.5.6 The Object Request Broker

The ORB component of CORBA provides more than just messaging mechanisms for remote object invocations. It also provides the environment for managing objects, advertising their presence, and describing their meta-data. A current CORBA 2.4 ORB [112] consists of several parts, as shown in Figure 2.4.

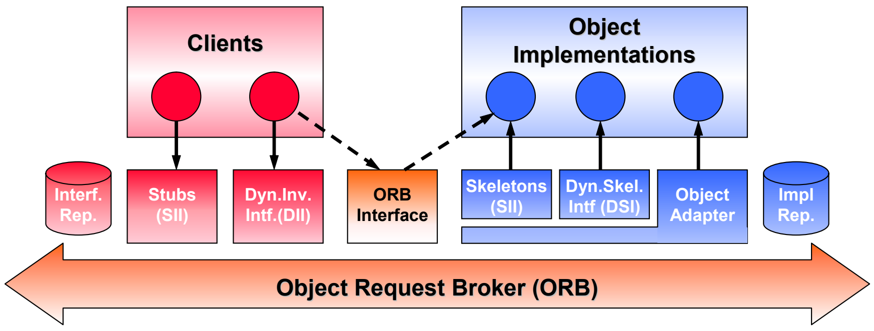


***Figure 2.4**: A CORBA Object Request Broker*

- The Interface Repository (IR) stores interface definitions. It allows the user to obtain and modify the description of component interfaces, the methods that they support, and the parameters that they require. The interface repository allows CORBA components to have self-describing interfaces.

- The client stubs – or Static Invocation Interface (SII) – provide an interface-specific API for invoking CORBA objects. A client application can invoke a server object through a client stub. From the client's perspective, the stub is a local proxy for a remote server object. Stubs are generated by an IDL compiler.

- The Dynamic Invocation Interface (DII) allows the creation of requests and invocation of objects at runtime. CORBA defines APIs for looking up the server interfaces, creating requests, generating parameters, issuing the remote call, and getting back the results.

- The ORB interface defines a few general APIs to local ORB services.

- The server skeletons – or Static Skeleton Interface (SSI) – provide static interfaces to server objects. They contain the code necessary to dispatch a request to the appropriate method. Skeletons are generated by an IDL compiler.

- The Dynamic Skeleton Interface (DSI) provides a runtime binding mechanism for objects that need to handle requests for interfaces not known at compile time. The DSI is the server equivalent of the DII.

- The Object Adapter (OA) provides the mechanisms for instantiating server objects, passing requests to them, and assigning them object references. A standard object adapter called the Basic Object Adapter (BOA) must be supported by each ORB implementation. From CORBA 2.2 on the specification introduces a new Portable Object Adapter (POA) that overcomes many of the BOA limitations.

- The Implementation Repository stores ORB-specific details about object implementations, their activation policy, and their identity.

3 Zero Copy Implementation Strategies

Previous work resulting from the early projects in network computing states the observation that badly designed I/O buses and slow memory systems in PCs or workstations are the major limiting factor in achieving sustainable inter-node communication at Gigabit/s speeds [90]. Today's high speed networks, I/O systems and hierarchical memory subsystems operate at comparable bandwidth of about 100 MByte/s. Therefore one of the most important challenges for communication system software is to *prevent data copies* in memory when communicating at full speed.

In this chapter we will first look at the question of whether the communication granularity matters and investigate for which granularities communication in commodity clusters can be improved and made fully efficient. Then the chapter gives an overview of why zero-copy software techniques are needed and of the software layers involved in extremely fast communication of user-level applications. Middleware is used to ease application development and is especially useful dealing with the additional complexity of software in parallel and distributed systems. The drawback is that the middleware introduces a further system layer which has to be studied and carefully designed to provide the expected performance. We explain in this introduction chapter why we selected CORBA over MPI as our a middleware of choice to study zero-copy optimizations.

3.1 Zero-Copy Communication

This dissertation argues that the in-memory copies are a bottleneck in communication system software. This bottleneck can be eliminated without special hardware, without changing protocols or interfaces and without compromising the protection mechanism of the user/kernel boundary. More importantly, this gain can be achieved in most cases without compromising the richness and variety of the creative style of programming that is possible only at the user-process level. Full-speed access to the fastest network devices in fact can be offered through standard system interfaces and for standard networking protocols.

Two key points must be considered when establishing a strategy for high-performance I/O. First, in a system where the memory copy speed is similar to, or even lower, than the speed of the I/O bus or the network, performing a single copy of data during I/O is unacceptable. The streaming performance of modern memory systems is normally slower than the I/O busses and even slower than the network. Fast caches do not help since large amount of data are to be transferred and there is rarely any immediate reuse of data.

The second point that must be considered is that most I/O interface architectures require that data is staged in main memory before it is consumed by a user level process which will read it. For example, the high performance of many filesystems depends on read-ahead caching, where data is optimistically placed in memory before a process requests it. Such a strategy permits that a following request may be quickly satisfied. This observation also holds for networked filesystems. Different message passing paradigms allow to send data to a receiver without waiting for a corresponding receive operation of the destination process. This is certainly the case for e.g. full postal semantics where the data must be received and stored in buffers by the operating system or the communication library. For commodity hardware where protocol processing is performed by the main CPU, one is faced by the same situation. Packets have to be received by the system before knowing to what communication stream they belong to and buffering becomes inevitable. This leads to the suboptimal performance of TCP over Gigabit Ethernet as shown in Figure 3.1. An optimized hardware with special purpose interfaces on the other hand can achieve real Gigabit/s throughputs as seen with Linux-BIP on Myrinet. The figure is further explained in Section 3.3.1.

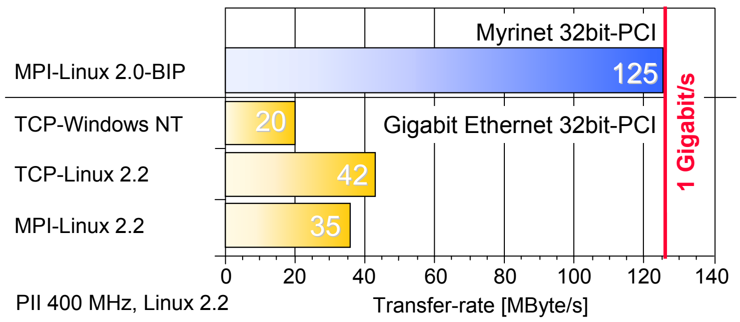


***Figure 3.1**: The operating system software cannot keep up with the hardware speed for Gigabit Ethernet and TCP/IP whereas special purpose hardware allows real Gigabit/s communication even on simple commodity PCs.*

To obtain the good performance of BIP with Myrinet it is therefore necessary to eliminate all in-memory copying, and to allow true zero-copy communication for all data that arrives in memory from the network before it is requested.

This dissertation describes the design, implementation, and performance of a zero-copy communication environment without weakening the protection mechanism of the user/kernel boundary or placing semantic restrictions on the application programming interface nor restrictions on the types of network devices that can be used. The fundamental techniques that are employed to achieve this is virtual memory remapping in combination with protocol speculation and a separation of control and data transfers. The memory remapping technique moves data in memory between the address spaces of the process and the kernel, without copying the data. This data exists in buffers kept by the kernel or the user that are just remappable memory pages. The newly introduced technique of protocol speculation allows to communicate these memory pages across interconnects that do not provide reliable connections in hardware and with a maximal transfer unit (MTU) smaller than such a memory page. This is required to use the ubiquitous Ethernet technology with zero-copy. The software design enables the benefits of copy-free virtual memory remapping for all forms of networking I/O, and also enables additional optimizations that further improve I/O performance. On top of more efficient communication libraries or more efficient frameworks for object oriented distributed computing like CORBA the user applications can be optimized for development effectiveness and for communication efficiency and high bandwidths. While we keep all API (application programming interface) standards we introduce separate primitives for control and data transfers in the ORB implementation.

3.2 Communication Granularity

The viability of a zero-copy technique often depends on communication granularity. In this section we will compare some PCI based networking systems to the supercomputer communication system of the T3D to find the native modes of communication with PCI systems and to determine the right level of data granularity where zero-copy communication can still make sense on commodity systems. We therefore rely on three readily available networking technologies for Intel based systems to study this issue: Myrinet, SCI and Gigabit Ethernet which were introduced and characterized in Section 2.2.

In addition to transferring contiguous blocks which is the natural way to communicate in message passing systems, the Direct Deposit Model [145] allows to copy fine grained data directly out of the users data structure at the sender into the users data structure at the receiver, involving complicated access patterns like indexing or strides. Old supercomputers could handle fine grain communication well unlike the now modern commodity systems which fail at fine grain patterns.

In most applications we encounter a few very characteristic communication patterns when data arrays are partitioned and redistributed. For transposes of distributed arrays and many other redistributions, every processor must exchange data with every other processor i.e. perform an “all-to-all personalized communication” (AAPC) operation. The end-to-end performance of an array transpose is largely determined by the ability of the memory and communication system to collaborate and handle local and remote copy

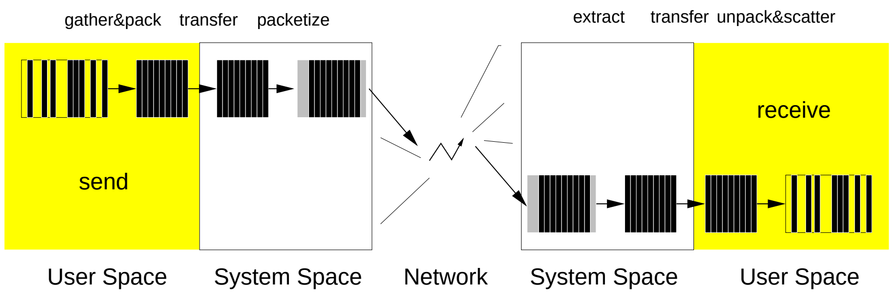


***Figure 3.2**: Reference model of basic communication steps for traditional message passing libraries including extra steps within the application or middleware for packing and unpacking of the message data at both endpoints.*

transfers with strides (dense matrices) or indices (sparse matrices) optimally.

Figure 3.2 shows a typical scenario, and illustrates the different steps of traditional message passing implementations as they are performed for a typical parallel computation. In practice, the data to be exchanged are usually stored in program variables, and those are not generally stored in contiguous memory locations. Therefore for a fine-grained communication step normally looks like this: The sender first gathers and packs the data from its original location into contiguous blocks for more efficient transfers, say a strided access to an array it wants to communicate. This is then made available or transfered to the system space. The send function may complete at this time (for the sender application), and the user may or may not be free to overwrite the data. Then network protocol headers, trailers and routing headers are generated which wrap the message. This step potentially breaks large messages into smaller packets and fragments the message. All outgoing message packets are queued and injected into the network if the network is ready to accept another message. On the receiver this packet is correspondingly received, the data extracted from the wrappers and the packets are reassembled into messages if necessary. Incoming messages are buffered or matched with pending receives. Then the data is transferred from system space to user space, the internal buffers are released and the data scattered to its final location by the application or the middleware.

As we will see in our evaluation different communication technologies have a different amount of hardware support for such transfers. This fact is exposed by our study of deposit transfers explained in this section.

3.2.1 Communication System Comparison

The key functions of a communication system within a distributed system is to move data and to provide explicit synchronization for consistency. This can be done at different abstraction levels with more or less support by the system. We attempt to find a common denominator for an evaluation and the comparison of different Gigabit interconnects by selecting a few common operations and examine the ways those can be performed with different abstractions and different amount of support from system software. At the lower levels the performance results are highly transparent and we can easily relate them to the specifications of the hardware while at the higher level the performance figures correspond closely to what an application can expect from the system. Therefore we propose to do the comparison at three different levels:

- **Direct Deposit:** for simple remote load/store operations. The performance at this level is expected to be closest to the actual hardware performance.
- **MPI Message Passing:** for an optimized standard message passing library. Carefully coded parallel applications are expected to see the performance measured at this level.
- **TCP/IP:** for a connection oriented TCP/IP LAN emulation protocol. Users that substitute a Gigabit network for a conventional LAN will see a performance comparable to this benchmark.

Direct Deposit with its simple “no fuzz” remote store semantics allows us to evaluate the fraction of the hardware performance that is sustainable by software in the different technologies. The transfer of non-contiguous blocks exposes the capability of the hardware to handle fine grained communication. If the hardware is unable to execute fine grained transfers efficiently, aggregating copies must be used which might largely affect the sustainable bandwidths.

At a higher level we further explore the transfer modes of MPI message passing communication with full buffering according to clean postal semantics and alternatively with some common “zero-copy” shortcuts that typically restrict the semantics of the messaging API and that makes buffering unnecessary.

To investigate the performance delivered in a classic, connection oriented networking protocol required to provide reliable communication for some interconnects we picked the “TCP/IP over LAN” protocol or protocol emulation respectively.

In the latter two cases the transfer of contiguous blocks satisfies the needs of the data representations in the principle API, but additional copies of data may be unavoidable due to the tricky postal semantics of send and receive calls in message passing or due to the requirement for retransmission to achieve reliable connections in TCP/IP.

Direct Deposit

Conventional message passing programs use the same mechanism (messages) for control and data transfers. The Direct Deposit Model [145] classifies control and data messages based on their contents. Control messages are linked to synchronization and are often part of the protocol. Data messages contain a significant amount of data that is moved between nodes and benefit for better communication bandwidth. Control messages are mostly empty and low latency is all that counts for handling them efficiently. Most high performance interconnects permit low latency signaling for control messages. The corresponding issues of their implementation and optimization is beyond the scope of our evaluation in this paper. The major concern is with the data transfers. Typically the source of data is in the virtual memory of a user process on the sender node, and the destination is in the user's memory at the receiver.

In the deposit model only the sender actively participates in the data transfer, "dropping" the data directly into the address space of the receiver, without active participation of the receiver process. The model allows a clean separation of control and data messages. In the deposit model, control messages, barriers or semaphores are used to deal with explicit synchronization, and data messages are sent only when the receiver has signaled its willingness to accept them.

In addition to transferring contiguous blocks the deposit model allows to copy fine grained data directly out of the users data structure at the sender into the users data structure at the receiver, involving complicated access patterns like indexing or strides. As we will see in our evaluation different communication technologies have a different amount of hardware support for such transfers. This fact is exposed by our study of deposit transfers.

The deposit transfers can be implemented in software e.g. on top of an active message layer [158], where the sender node just sends the data, and a handler is invoked on the receiver to move the data to its final destination. However, our understanding of direct deposit suggests that a general control transfer in the form of an RPC should be avoided and that previously asserted synchronization is sufficient to move the data. Furthermore the functionality of the handler is fixed and the deposit operation at the receiver only affects the memory system of the receiver. Unlike in the original active messages a good and efficient implementation of direct deposit therefore mandates that the deposit handler is implemented directly in hardware, e.g. by DMAs or alternatively by a second "communication" co-processor, which executes only handlers and unpacks messages.

Direct deposit strongly resembles a simple remote store on a NUMA architecture. The difference between deposit and NUMA stores is that the deposit model promotes and assumes aggregation despite the non- contiguous access patterns that occur when communication data is placed directly to its destination in user space.

MPI Message Passing

The most widespread use of the MPI API is an example of the classical postal model. Both the sender and receiver participate in a message exchange. The sender performs a send operation and the receiver issues a receive operation. These operations can be invoked in

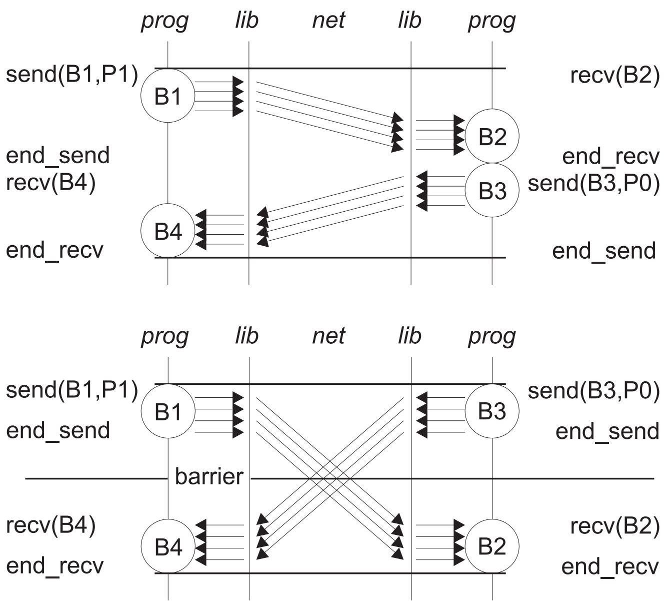


***Figure 3.3**: Two scenarios: First with restricted postal semantics based on synchronous, blocking send calls, which may release the sender only when the receive is executed; Second with full postal semantics where non-blocking calls are always permitted, forcing the communication system to buffer the messages until received.*

either order, blocking or non-blocking. That is, messages can be sent at any time without waiting for the receiver. This enhancement in functionality forces the system to buffer the data until the receiver accepts it.

An optimization, so that no additional copy is needed on the receiver node, delays the transfer and waits until the receive is posted which then pulls the data from the sender. We call this restricted postal semantics. Figure 3.3 shows two possible scenarios of postal semantics. The upper chart shows how data can be transferred directly when restrictions on the use of the send and receive calls are accepted. In the picture the restricted calls are synchronous and mutually block to wait for completion at both ends of the transfer. Restricted postal semantics is enough for this scenario and a message passing library can prevent data buffering.

In the lower chart full postal semantics is assumed and required. The messages can be sent without looking at the receiver. But they have to be buffered by the communication system until the receiver is eventually ready to accept them. This buffering might introduce additional copying of the data.

MPI-communication functions must be implemented with the lower level primitives that are offered directly by the communication technology or their software interfaces. Often this primitives are just remote load or remote store but it can be a socket API with a TCP/IP stack underneath too.

For clean higher level abstractions buffering is an important part of the message passing system. The typical amount of data transferred is usually too large to be stored in special purpose registers of the network interface or in the buffers along the path from a sender to the receiver. Therefore buffering is done by a higher level of the message passing library and involves the memory system at the end-points. Many simple libraries just allow to map the implementation of blocking sends directly to a fast direct deposit including synchronization functionality. Those can be compared with the restricted semantics scenario. The proper execution of non-blocking sends need buffering on the receiver side and often leads to an additional copy operation which again largely affects the performance. This mode of operation is characterized by the full postal message passing scenario. In a serious performance characterization of any MPI implementation both cases should be evaluated separately.

TCP/IP

Connection oriented LAN network protocols are particularly important for Clusters of PCs and crucial to their commercial viability. Most protocol stacks are provided by the default operating system of the PCs and many software packages using these protocols and the socket interface are available. In our Clusters of PCs (CoPs) project we do not want to limit ourselves to deploying ever cheaper GigaFlops to our research colleagues in computational chemistry or computational biology, but intend to widen the range of parallelizable applications from scientific codes to databases and Internet servers. Especially for web-servers or commercial databases and middleware-systems it would not be viable to change the standard communication protocols to restricted high speed messaging. For network file systems on clusters, like NFS or Sprite, both UDP/IP and TCP/IP must be provided. With an optimized IP or socket API implementation a Cluster of PCs can provide high performance at a good price for a larger number of programs, than a dedicated cluster with just a message passing system software does. Additionally networking protocols ensuring reliable transfers are crucial for gigabit technologies that do not offer reliable services in hardware.

IP is primarily designed for Internet communication and not for messaging in parallel systems. However it can offer an unreliable, connectionless network service fragmenting packets in IP-datagram and deliver them according to the IP address scheme. Transport protocols as UDP and TCP allow to extend communication to different processes of the same end system by a port concept (sockets). TCP further enables full duplex communication over a reliable data stream by implementing flow control and retransmission with a sliding windows protocol. The latter functions of TCP are less important in a cluster interconnect (there should be no loss in the switches) but its API is very common if not ubiquitous.

Because the protocol is implemented without specific knowledge of the used hardware, assuming an unreliable network service like Ethernet or the Internet respectively, the performance of IP will rarely match the performance of optimized MPI and direct deposit protocols. Especially the latency for TCP data transfers is much higher due to connection setup, which might be acceptable or unacceptable to certain applications.

3.2.2 Transfer Modes and their Natural Implementation

We focus on the performance of moving data itself and disregard any difference in amount of local or global cache coherency that the different technologies offer, since at this point none of the technologies can offer automatic fully coherent shared memory to support a standard shared memory programming in hardware. Such will remain a privilege of much less scalable or more expensive systems like bus based SMPs and directory based CC-NUMAs. Furthermore the implication of different network topologies is an extremely well researched topic and therefore we assume that a sufficient number of switches is used in both systems to provide full bisectional bandwidth as this is the case in most smaller systems. It is also clear the data transfers to remote memory must be pipelined and aggregated into large messages. The pure ping-pong latency of a single word transfer remains of little interest for a comparison of the sustainable end-to-end throughput achieved for different communication patterns in different transfer modes with processors, co-processors and DMAs all working.

Until recently the maximal performance of the memory- and the I/O-system was rarely achieved by the network interconnects. Therefore neither the performance of the I/O bus designs nor the performance of the common system software was optimized enough to work with Gigabit networking. Those two factors are the principle bottleneck in today's Clusters of PC's.

A further bottleneck is the lack of local memory system performance in PCs. Memory system performance is not only important to computational efficiency in applications with large datasets, but it is also the key to good performance of inter-node communication. While high end MPP node designs can afford memory systems with special hooks for inter-node communication at Gigabit/s speeds, low end systems must rely entirely on mass market memory systems and standard I/O interfaces (i.e. a PCI bus) for economic reasons.

The main difference between different adapters of different network technologies are the default and alternate transfer modes they can operate in. Although the hardware mechanisms involved in transfers between main memory, staging memory and network FIFO queues may be vastly different, the purpose of all data transfers remains the same: Move data from user space of a sending process to the user space of a receiving process. Most interconnect designs can do this with close to peak speed for large blocks of data and for the special semantics of zero-copy messaging.

To explore these issues in more depth, we require that a direct remote memory operation can also include more complex memory operations at the receiving end e.g. strided stores. A typical application for such an operation would be the boundary exchange of an iterative FEM solver working on large, distributed sparse matrices.

We study two options for transfers for each of the interconnects discussed. One mode is mostly processor driven and utilizes the most direct path from memory to the network FIFOs, while the other mode is DMA driven. The latter mode makes a few additional copies on the way to the network interface but uses DMAs to do them in parallel to the regular activities of the processor.

To implement the remote memory system performance tests with their complicated strided patterns we can either use the hardware support provided by the communication adapter or the packing/unpacking with the main processor that always leads to an additional copy operation, but avoids the inefficiency of single word transfers across the PCI bus.

- **Direct deposit, mapped**: The main processor pushes the data directly into the network FIFOs through regular store operations addressed to a special segment of virtual memory, mapped directly to the network interface and through that port on to the memory of the remote processor. Contrary to a common belief the precise layout of the assembly instructions to trigger this remote store or load operation does not matter. It is well conceivable that a parallelizing compiler handles a remote store as two separate stores for address and data, since the compilers have to know about local and remote for performance optimizations.

  This mode of operation is the native mode of operation for an SCI adapter and for the Cray T3D which have both direct hardware support for it. A Myrinet adapter can only map the send registers and the staging memory (SRAM) directly into the user address space of the application, but not the remote memory itself. It seems that direct remote stores are impossible unless the two dedicated co-processors at the sender and the receiver side become involved. A dedicated control program for those co-processors must shadow the adapters staging memories, transfer the data across the network and move the incoming data to the remote memory at the receiving end, so an SCI like remote store operation can be emulated for contiguous and strided blocks of data. This technique does not work too well for an isolated store but performs adequately for an aggregation of multiple stores with indexes or strides. For Gigabit Ethernet, especially for simple standard interface cards without programmable processors, this mode can not be implemented in an efficient way. The only possibility would be to implement a descriptor list, which scatters the Ethernet packets according to the specified stride but unpacking by the processor is the much more natural way.

- **Direct deposit by DMA**: The application stores its data (and potentially also the addresses) into a reserved, pinned address segment of local memory instead of the mapped remote address space. Starting from there the DMA engine of an adapter can pull the data directly into the network FIFO queues on the interface for transmission.

  This mode of operation works very well for *contiguous* blocks of data. SCI in direct message passing mode can send blocks of the mapped memory using its DMA. The DMA controller utilizes the most efficient sequence of SCI transactions to achieve highest possible throughput. Myrinet with its three DMAs on the LANai allows a similar mode of operations. Furthermore there is a bit more flexibility since data can be gathered in small portions, stored at the staging memory and sent directly to the packet interface. The DMAs can be supervised and periodically restarted by the Lanai MCP. For Gigabit Ethernet this mode can be used assumed that the hardware can fragment the data and enclose the fragment by protocol headers and trailers. This technology is discussed in detail in Chapter 4.

- **Buffer packing with processor or DMA at the sender**: The main processor or a DMA of the PC gathers the segmented data into the network FIFO via a segment of mapped main memory. Then, the network card's co-processor or another DMA transfers the message into the network FIFO.

  In this mode the message is processed by either the main processor or a network processor and a DMA and finally transmitted into the network FIFO. Measurements indicate that this is the best way of transferring data over a PCI bus system and it can therefore be called the *native* sender mode for either Myrinet or Gigabit Ethernet. Depending on the block size of strided and indexed transfers packing with the main processor first can be faster than gathering them directly with the DMA.

- **Buffer unpacking with processor or DMA at the receiver**: The main processor packs the destination store addresses along with the data into a message. The adapter gets the prepared message by a DMA and pushes it into the network FIFO. On the receiver node the main processor or the network card's message processor reads address and data words and scatters the data via DMA transfers to a segment of main memory.

  Again the message is processed by either the main processor or a network processor and a DMA at the receiver side. Measurements indicate that this is the best way of transferring data with scattering on the receiver and can therefore be called the *native* receive mode for Myrinet and Gigabit Ethernet. Depending on the block size and access pattern unpacking with the main processor can be faster than scattering them with the DMA.

3.2.3 Performance Evaluation

In our performance evaluation we discuss the throughput to move data in different application scenarios. For the best understanding of our measurements it is important to keep in mind at which level the benchmark is performed (lowest level of deposit or highest level of LAN emulation) and which data path (mode of operation) is used.

For all modes of operation that involve packing/unpacking operations of messages or buffers, the local memory system performance is very important for communication performance. As a preliminary step to our evaluation we developed a novel memory system micro-benchmark. Unlike the simplistic McCalpin loops [102] our test captures all aspects of the memory hierarchy, in particular its performance behavior with temporal- and spatial locality (varying working sets and strides) [142, 143, 93]. Our ECT memperf (Extended Copy Transfer Characterization) benchmark [92] therefore goes beyond pure loads and stores bandwidth to measure the copy throughput with a simultaneous load and store data stream. Most end-to-end transfers in compiled parallel programs involve fine grain accesses, either with strides into arrays or a large number indexed smaller blocks to gather/scatter data from/into distributed collections of objects. Therefore we graph access patterns for various strides, from contiguous blocks up to single 64bit entities spaced a constant strides of 64 apart.

The same method is used to measure remote accesses for machines with full, partial or without support for coherent shared memory. For the measurement we use the framework for characterizing local and remote memory system performance [142].

For the best characterization of an interconnect technology, the remote memory system performance figures for different access patterns (strides) and a large working set are the interesting issue. The measured copy performance for the same operation in the local memory system is included into the Figures 3.4 - 3.6 just for a comparison. Since we are only discussing remote deposit (and leave out remote fetch) all transfers in our charts are done by contiguous loads from local memory and strided stores to the same local or remote memory system.

To approximately match the performance of the T3D Alpha system with the commodity PC systems we chose a 200 MHz Intel Pentium Pro system with an Intel 440FX Motherboard Chipset. The processor, the data bus to SDRAM and the PCI Bridge are connected by a proprietary 64-bit 66 MHz host bus (512 MByte/s). The external PCI bus is 32 bit wide and runs at 33 MHz (132 MByte/s).

The performance numbers in Figures 3.4 - 3.6 show, that the copy bandwidth of the memory system on Pentium Pro PC's of only 45 MByte/s is much less than the peak performance of modern interconnects. Copying data in the same memory system with the processor is therefore always a bottleneck and must be avoided at all cost.

Performance of Direct Transfers

The performance of direct transfers is measured for contiguous blocks (stride 1) and for increasing strides (2..64). The first set of performance curves in Figures 3.4 - 3.6 (bullet) marks the performance for the most direct transfers by store operations to the mapped remote memory or an emulation thereof. The second curve (triangle) marks the performance for highly optimized buffer packing transfers. In this case the transfers were optimized as well as we could. If the DMA was faster, then DMA was used. The relationship between local and remote memory performance can be understood by comparing the performance curves of local memory for the corresponding copy operation to the direct and buffer-packing remote performance curves (squares).

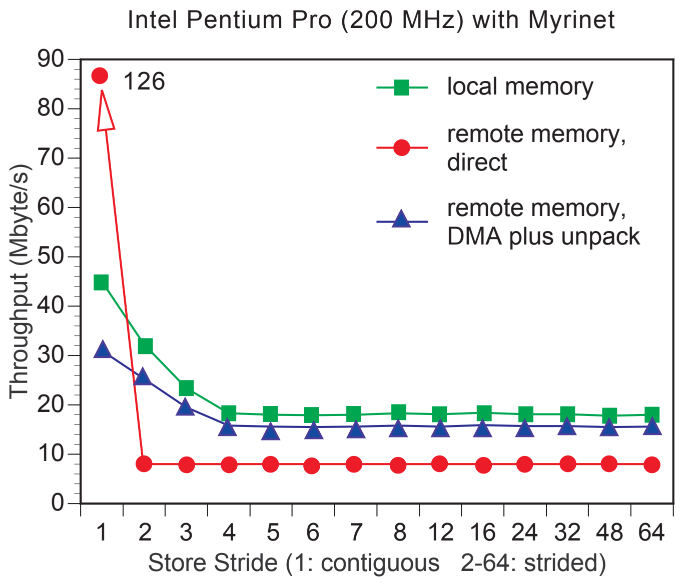


***Figure 3.4**: Throughput of the Myrinet Host Adapter for direct mapped emulation and buffer packing transfers. As a reference the performance of the local memory system is given for the same copy operation (contiguous loads / strided stores).*

For Myrinet (see Figures 3.4) the store by the DMA is very much affected by the size of the chunks transferred by one DMA activation. The buffer packing mode with the main processor seems to perform at about memory copy bandwidth whereas the DMA transfers suffer from the overhead of too many DMA initializations and too many PCI bus arbitrations. The buffer packing, native mode can fully use the DMAs to boost the case of large contiguous blocks.

For SCI we also observe good performance for contiguous blocks in direct transfer mode (see Figure 3.5). This is when the eight stream buffers work optimally. For strided data the performance of remote stores on PCI collapses to under 10 MByte/s and appears

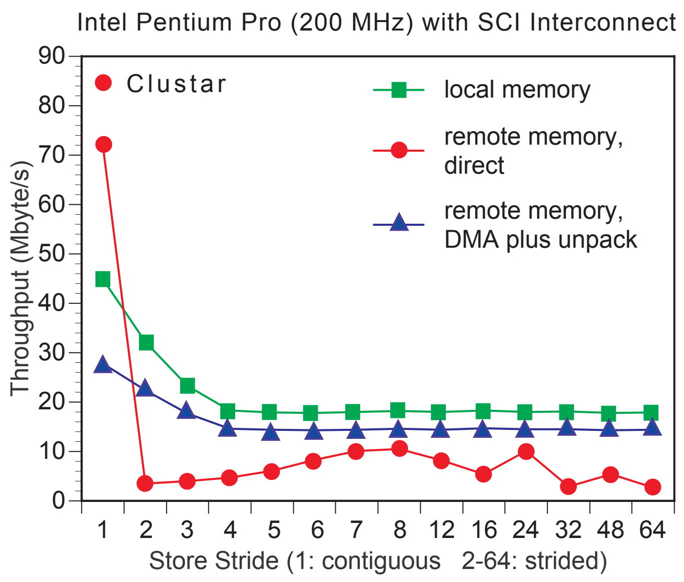


***Figure 3.5**: Throughput of the Dolphin PCI SCI Adapter for direct mapped and buffer packing transfers. As a reference the performance of the local memory system is given for the same copy operation (contiguous loads / strided stores).*

to be unstable. The sloped curve from stride 2 to stride 8 can be explained by the mechanics of the stream buffers. For stride 2 only one stream buffer is used as the direct mapping of the stride 2 addresses to the same 64 Byte stream buffer leads a sequence of non-pipelined single word transfer. The buffer is always sent directly with only 8 Bytes of data and the next value addressed to the same buffer has to wait, until an acknowledge releases the buffer again.

As with Myrinet it turns out to be faster with SCI to unpack a communication buffer than to execute transfers directly in this case.

The Cray T3D on the other hand offers (or better offered) a completely different performance picture (see Figure 3.6). It turned out that it was always best to execute a data transfer in direct mode. Buffer packing included copies and those slowed down the transfers. For contiguous blocks a direct copy to remote memory was even faster than a local copy from and to memory - this is not surprising since two memory systems - one at the sender side and one at the receiver's side are involved for a single data transfer in the remote case.

We did not measure direct transfers with Gigabit Ethernet, as this traditional wide area technology relies on a TCP/IP protocol stack to ensure reliable transfers. The performance that can be achieved in maximum is limited by the copy bandwidth of the memory system.

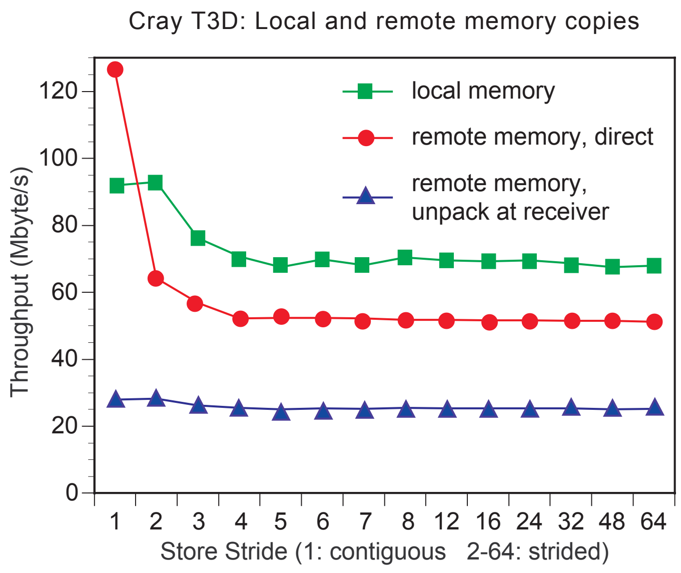


***Figure 3.6**: Throughput of the Cray T3D for direct mapped and buffer packing transfers. As a reference the performance of the local memory system is given for the same copy operation (contiguous loads / strided stores). Note that two T3D nodes can exchange data faster than a single node can copy it.*

Performance of MPI Transfers

The performance of the higher level transfers indicates how well system programmers can work with the hardware. We used a full function standard message passing libraries with buffering for true postal message passing and a reduced zero-copy library for reasons explained in the earlier section. So our evaluations use two different tests exposing the performance at full postal functionality with buffering and at the reduced functionality with direct transfers.

On Myrinet we use BIP-MPI [129] for tests in message passing. BIP (Basic Interface for Parallelism) is a high performance library with a possibility simplified data transfer semantics. The code was developed at the ENS of Lyon, France. Although this implementation might eventually fall a bit short in terms of compliance with the extensive MPI standard, it seems to provide a stable API for all important basic functions. BIP-MPI is a modified MPICH version using BIP to drive the Myrinet network hardware. The performance of BIP-MPI (see Figure 3.7) matches the raw performance at 126 MByte/s for blocking send and receives measured with large blocks ($>$ 1 MByte). Half of peak performance can be reached with messages of roughly 8 KByte size.

For SCI we we used the ScaMPI MPI implementation [9]. This is a fully standardized version delivered by a joint research project between industry and academia. The same picture as with Myrinet is given for SCI. For blocking send and receives measured with large blocks good performance can be achieved, but copying prevents better results

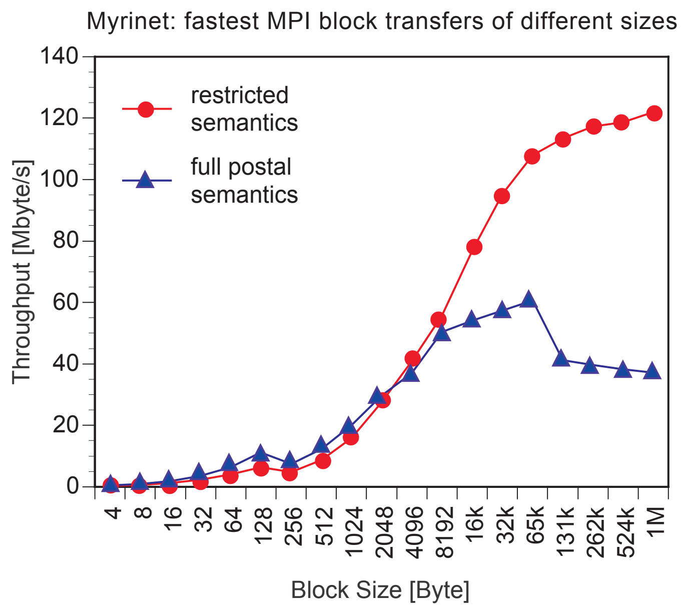


***Figure 3.7**: Throughput of BIP-MPI transfers over Myrinet either for restricted and full postal semantics.*

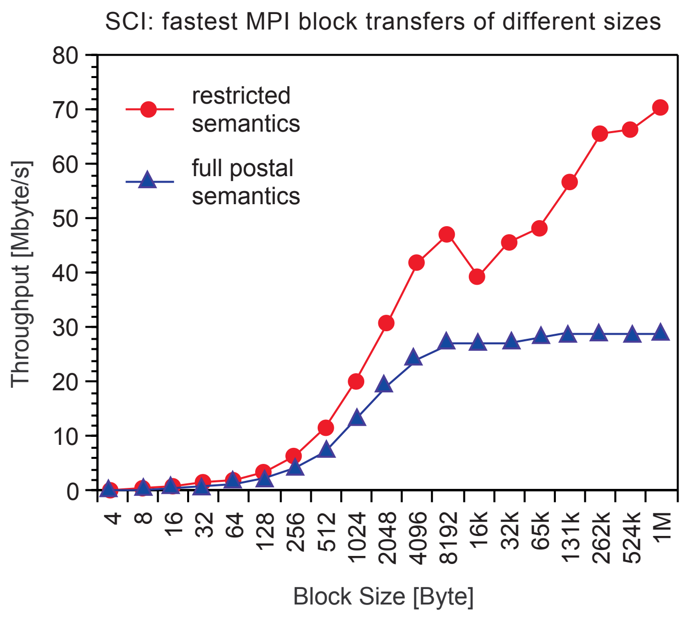


***Figure 3.8**: Throughput of ScaMPI transfers with SCI either for restricted and full postal semantics.*

for the non-blocking case. Here again the performance drops to the local memory copy bandwidth for strided stores.

For the non-blocking calls, where the sends are posted before the receives, MPICH enforces buffering which drops the performance to the one of the local memory copy. An optimization uses the LANai staging memory wherefore blocks until 64KByte can be buffered without an additional local memory copy. When two sends are posted before the receive the peak performance is measured at 32KByte blocks.

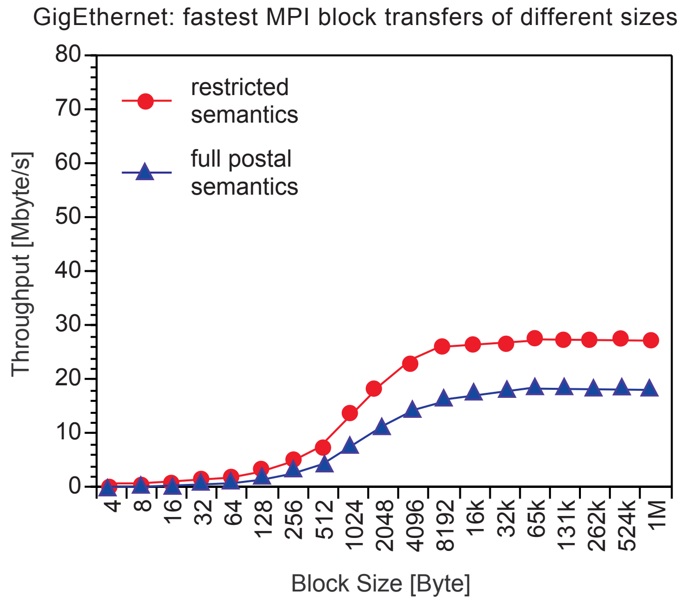


***Figure 3.9**: Throughput of MPICH transfers with Gigabit Ethernet either for restricted and full postal semantics.*

For Gigabit Ethernet again the bandwidth largely relies on the copy bandwidth of the memory system as all the MPI transfers are mapped to TCP/IP transfers. Even with restricted semantic and without buffering in the communication library, copying is involved by the communication stack. We used the MPICH implementation using TCP/IP as transport protocol for the measurements with Gigabit Ethernet. The results are very unsatisfactory but not surprising. The TCP stack introduces a memory copy, so that even the reduced semantic which does not need buffering in MPI results in bad performance of just 30 MByte/s. Like with the Myrinet MPICH implementation, the non-blocking calls, where the sends are posted before the receives, enforces buffering which drops the performance again because a second local memory copy has to be performed in the library resulting at 22 MByte/s.

For the Cray T3D we show results of a high performance MPI implementation for the by the Edinburgh Parallel Computing Center in cooperation with Cray Research (CRI) [24] (see Figure 3.10). The EPCC MPI provides the full MPI specification and was developed using the SHMEM get/put primitives on the T3D. The measured performance

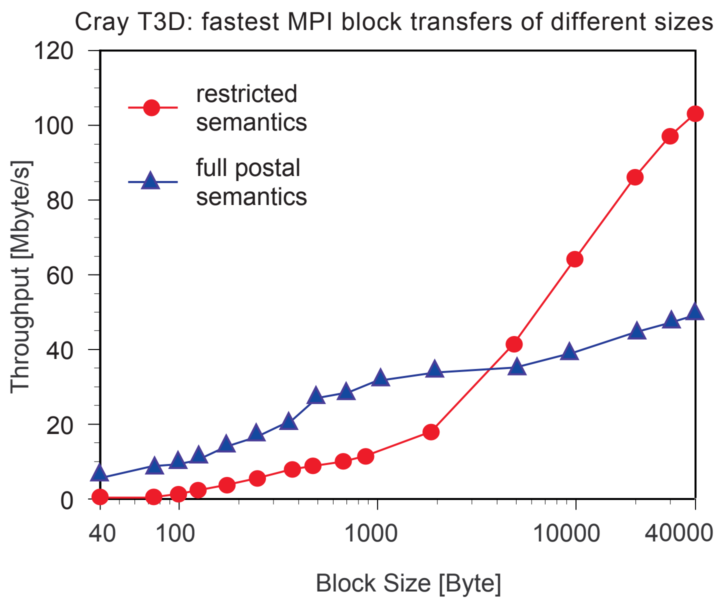


***Figure 3.10**: Throughput of EPCC-MPI transfers for the T3D either for restricted and full postal semantics.*

corresponds to the BIP-MPI over Myrinet where blocking calls don't need any buffering whereas non-blocking semantics slows down by an additional copy in the local memory system.

Performance of TCP/IP Transfers

For a significant growth of market share for clusters of PCs it is most important to port traditional applications quickly and easily by simply substituting a conventional LAN network (e.g. switched 100BaseT) by a gigabit technology interconnect, especially in the booming area of servers for the Internet and for the important application of distributed database and middle ware systems. In this mode of operation the performance of a fully standardized protocol stack like TCP/IP is essential. Therefore we measured the standard and the best LAN/IP emulation packages that were available for each interconnect technology (with the exception of the T3D, where IP does not make much sense).

The Linux TCP/IP stack is a one copy implementation that has to fragment and defragment the user data passed to the communication socket. After the fragmentation, the protocol data is generated and and send to the NIC hardware. On the receiver the data is must be defragmented and copied to the receiver process. This explains why the TCP/IP performance for Gigabit Ethernet can not be faster than a local memory copy. The protocol overhead does the rest and a poor performance of 33 MByte/s can be achieved for large contiguous block transfers.

Myricom offers TCP/IP emulation by a fully compliant TCP/IP protocol stack that transfers data at 20 MByte/s. BIP-TCP [129] improved this performance by using the "zero-copy" BIP interface so that about 40 MByte/s are reached. The BIP implementers use the original Linux protocol stack and substituted the transfer mechanism using their BIP message passing system for the lower layers.

For our SCI cards we could only reproduce 22 MByte/s with a TCP/IP protocol stack implementation similar to the BIP-TCP of PC2 Paderborn. It uses the standard Linux stack and substituted the transfer mechanism using remote mapped segments.

3.2.4 Communication Granularity Achievable in Commodity Clusters

The above study clearly shows that excellent performance for direct remote memory operations on low-end Gigabit/s technologies with PCI-based systems is very limited to a few ideal cases. Except for Gigabit Ethernet, the transfer rates typically peak near the bandwidth limits of the PCI-bus or the interconnect and are almost comparable to the rates seen in traditional MPP supercomputers.

But such good performance is only achieved for the most simple transfer modes like direct remote deposits of contiguous blocks of data and remain restricted to non buffering MPI communication scenarios. For strided data accesses the performance of the SCI and Myrinet interconnect totally collapses, while the traditional MPP can do those cases at acceptable speeds. The DMA transfers on the commodity PCs suffer from the overhead of too many DMA initializations and too many PCI bus arbitrations which can be overcome in a buffer packing mode with the main processor limited by the local memory copy bandwidth. For Gigabit Ethernet there is no way in doing direct deposit in hardware and the performance of the buffer packing mode is similar to the cluster interconnects as also the local memory copy bandwidth is the limiting factor. For the implementation of message passing libraries with buffering semantics (e.g. MPI) the performance of the all interconnect is again reduced to the local memory system bandwidth while the traditional MPP can do those cases at much better speeds. Similar limitations due to copies in the local memory system slow down the IP over LAN performance in most cases. Especially with Gigabit Ethernet, which relies on a TCP/IP stack, just a small fraction of the hardware performance can be achieved, mostly due to memory copies in the operating system stack.

The performance for the special purpose hardware is remarkable given the low cost of these interconnect boards. It stays close to a Gigabit but due to the aforementioned limitations we expect that any Cluster of PCs built with any PCI card interconnect will get into some difficulties with applications that require complex, dense communication patterns or rely on standardized high level networking APIs and protocols such a TCP/IP. With such condition precedent to the communication the performance of fine grain accesses is always limited by the memory system of the host, as the data has to be packed and unpacked to be transferred.

Therewith we can state that communication *granularity matters* largely. Zero-copy transfers are an absolute prerequisite for large blast transfers where large bandwidths are requested. For dense communication patters, packaging and contiguous transfers are always faster even with dedicated remote shared memory hardware and zero-copy is therefore not worthwhile. We therefore concentrate on bandwidth sensitive applications and expect a fast memory system to be available for dense communication patterns.

3.3 Motivations for Zero-Copy

The discussion in the last section exposed the real bottlenecks in commodity high speed networking for clusters of PCs: The evident lack of performance for the Gigabit Ethernet technology. This lead to my investigations of whether this technology will be successful and can ever perform at levels interesting for cluster computing.

3.3.1 A Lesson about Software Inefficiency

In the meantime Gigabit Ethernet networking hardware became readily available and will be commodity and very cheap in the future (i.e. it already comes directly build on server motherboards and even with laptop computers). But the discrepancy between hardware performance and overall system performance remains the highest among the three examples discussed. Disappointing communication performance results if standard interfaces and protocols such as the socket API and TCP/IP are used, even when using special purpose technologies. Figure 3.11 shows the data rates achieved for large transfers with the French BIP MPI library [129, 64] for Myrinet and with standard TCP/IP protocol stacks as of fall 1999 for Gigabit Ethernet. The tests execute over the different technologies interconnecting the same PC hardware, expecially the same I/O-bus and the same memory system. Even if the tests use different middleware or communication libraries they implement the same task and the result is exactly identical. This implies that the state of the art in gigabit networking with commodity components at the border of the third millennium is still not what we would expect it to be.

The Myrinet-MPI performance is close to the PCI bus limit and proves that data transfers at a Gigabit/s speed can indeed be achieved even with commodity platforms using a PCI bus based network interface card. The TCP performance is about one third of the maximal achievable rate and shows the problem we are focusing on, the poor performance with a socket standard interface using TCP/IP and Gigabit Ethernet. One reason of the good performance of MPI over Myrinet are large packet sizes and that there is no need for packet fragmentation.

This fact really shows the need for new techniques especially for commodity Gigabit Ethernet and TCP/IP where the software cannot keep up with the hardware. Advances in network bandwidth follow a step function, but the fastest networks tend to stay close to

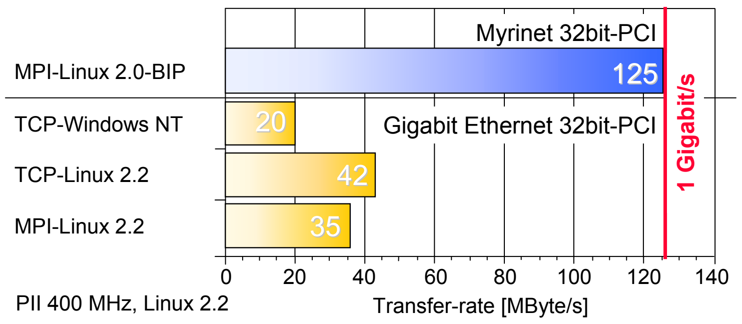


***Figure 3.11**: The throughput numbers for large data transfers over Gigabit/s point-to-point links with standard and special purpose networking technology and communication system software show that the operating system software cannot keep up with the hardware speed for Gigabit Ethernet and TCP/IP.*

the limit of the hosts. Given a sufficiently fast network, achievable TCP performance depends on optimizations to minimize networking overhead. With Gigabit Ethernet widely deployed and 10 GB/s Ethernet on the horizon, these optimizations are highly relevant again today, three years after the original experiment was conducted with the first Gigabit Ethernet hardware available.

3.3.2 Where Does the Efficiency Get Lost

Modern TCP/IP implementations can transfer data at a high percentage of available network link bandwidth, reflecting the success of many years of refinements [84]. Clark and Jacobson proved in [29] that protocol costs for TCP/IP are significant but they are not the limiting factor for a well implemented TCP/IP stack. Among other problems that were found, previous stack implementations used a strict layering model which caused bad buffering decisions and unnecessary memory traffic. Another important optimization proposed lead to protocol processing, that could be accelerated significantly by predicting the next arriving packet based on packets that have been previously received. Even though TCP needs to be able to handle many kinds of different circumstances, e.g. connection setup and packet loss, in most cases packets arrive in order and without errors in the data they carry. This standard code path should be optimized, even at the cost of making recovery from error conditions more expensive.

To find out the system factors that can limit bandwidth for TCP on high-speed networks we measured the overheads for TCP communication in a Linux system. Another detailed analysis is given in [85].

On the fastest networks, application-to-application throughput is limited by the capability of the end systems to generate, transmit, receive, and process the data at network speeds. Delivered performance is determined by a combination of factors relating to the

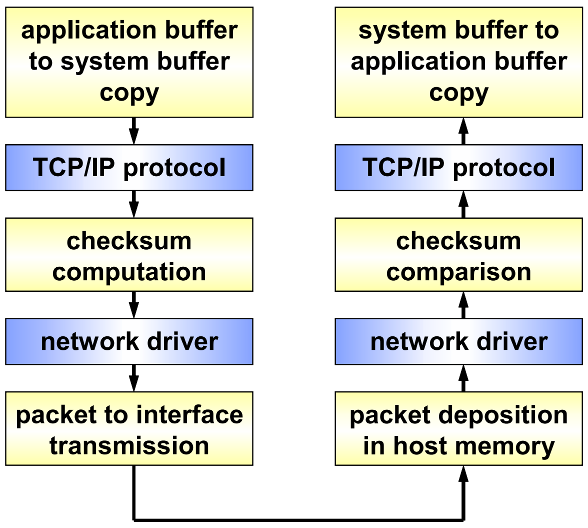

application buffer to system buffer copy
TCP/IP protocol
checksum computation
network driver
packet to interface transmission
system buffer to application buffer copy
TCP/IP protocol
checksum comparison
network driver
packet deposition in host memory

***Figure 3.12**: Sources of end-system overhead for TCP/IP*

host hardware, interactions between the host and the network adapter, and host system software.

Data transmission using TCP involves the operating system (OS) facilities for memory and process management, as well as the TCP/IP protocol stack and the network device and its driver. Figure 3.12 [27] identifies the key sources of overhead when transmitting data over a TCP connection in a typical system. CPU overhead for processing each network packet or frame occurs in the end points. These *per-packet costs* include the overhead to execute the TCP/IP protocol code, allocate and release memory buffers, and field device interrupts for packet arrival and transmit completion. In addition to the per packet costs the system is charged with costs for each byte of data sent or received. These *per-byte costs* incur at the stages shown in white in Figure 3.12 and include overheads to move data within the end system and to compute checksums to detect data corruption in the network.

The potential for high bandwidth has little value in practice if communication overheads leave no CPU power to the application to process the data. CPU utilization is just as important as bandwidth, since the bandwidth requirement of any application will drop if application processing alone saturates the CPU.

All the optimizations we propose in the thesis, are fundamentally directed at reducing overhead; they increase the delivered bandwidth indirectly by preventing the saturation of the host CPUs. The bandwidth measurements shown in Section 3.3.1 all occupied the CPU to nearly 100%.

To better understand the costs responsible for the CPU utilization, we carefully instrumented the Linux 2.2 TCP/IP stack and NIC driver with CPU performance counter profiling points with low overhead and measured the single overheads on the different operating system levels for a receiver at the given maximal bandwidth of 42 MByte/s.

Figure 3.13 shows the breakdown of receiver overhead into the following five categories:

- data movement overheads for copying and checksumming,
- virtual memory management costs (buffer page allocation and page remapping),
- TCP/IP protocol stack overheads,
- interrupt handling for received packets and
- driver overheads and DMA initializations.

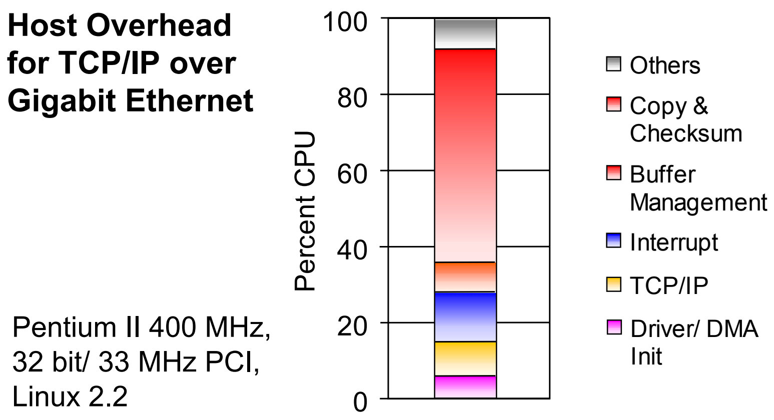


***Figure 3.13**: Host overhead for TCP-IP communication over Gigabit Ethernet on a PC running Linux at 400 MHz.*

The measurement was done for large data transfers with the maximum Ethernet packet size of 1500 Byte (Maximal Transfer Unit). The CPU utilization of nearly 95% at a bandwidth of 320 MBit/s explains the limited bandwidth achievable with this technology.

The graph shows that about 50% of CPU time is spent for some data movement at the socket layer. At the same time the checksum can be calculated and this is more or less for free as the data movement is responsible for the bottleneck. The less important overheads are the buffer management with 6% and the processing of interrupts with 13%. The TCP/IP stack overheads do not exceed 9% on a 400 MHz Pentium II which confirms the result described in [29].

Copying and checksumming optimizations are extremely important even on the platforms that are strong enough to achieve peak bandwidth without optimizations. Any reduction in overhead translates directly into lower CPU utilization, leaving more cycles available for a higher packet volume leading directly to higher bandwidth or higher application processing rates at a given bandwidth.

It is tempting to assume that the advances in CPU power given by Moore's Law will render these limitations increasingly irrelevant, but this is not the case. The limiting factor is not CPU processing power but the ability to move data through the host I/O- and memory systems. Wider data paths can improve raw hardware bandwidth, but more bandwidth is invariably more expensive for a given level of technology.

3.3.3 Trends in Memory and Network Speed

CPU clock cycle speeds, together with the number of instructions that can be issued in one cycle, determine the rate at which processing can be performed. I/O operations require processing at several levels, such as the manipulation of the representation of data as it moves through the system. Memory speed affects the rate at which data can be moved to, from, or within memory. I/O data typically resides in at least one location in memory while some I/O operation is performed.

CPU and memory speeds show a technology trend that directly relates to the performance of an I/O subsystem. Although CPU and memory are the two most tightly coupled components of a computer system, they have completely different curves of advancement in technology (see Figure 2.1 in Section 2.3.1).

This also holds for advances in network speed over time. Figure 3.14 compares memory speed versus network speed improvements in four to five years. While the memory copy transfer bandwidth improved by a factor of 4 from 1996–2000 the commodity Ethernet network bandwidth got 10 times faster in the same time period, and for the near future 10Gig Ethernet is on the horizon. Memory system speed improves steadily, but a trend that network bandwidth matches or even exceeds the increase in memory copy bandwidth is clearly visible.

Looking at this performance picture we can solely argue that zero-copy techniques will be essential to optimal system performance in the future.

3.4 Layered Systems

Complex layered systems, and even complex monolithic systems, can always be classified as a hierarchy of levels of abstractions. The decomposition of complex monolithic systems into subsystems or layers can improve the view and understanding substantially.

The idea of near-complete decomposability introduced and studied by Ando and Simon 1961 [138] in the field of economic structures and in a variety of biological models has been brought to computer science by Courtois [32, 33]. His important observation was that large computer systems can usefully be considered as nearly completely decomposable systems - systems arranged in a hierarchy of components and subcomponents with strong and fast interactions within components compared to weak or slow interactions between these components.

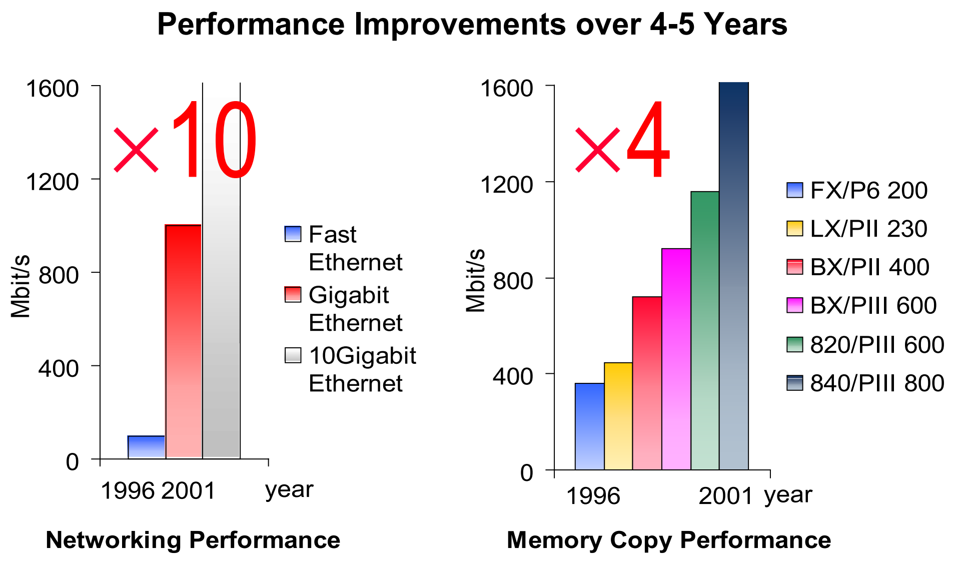


***Figure 3.14**: Speed improvements of network copy bandwidth for commodity networks (Ethernet) and memory copy bandwidth in commodity PCs from 1996 to 2001. Networking speed come close to memory speed and improve faster, memory system bandwidth improves more steadily instead.*

Decomposability in program behavior has been used by Dijkstra for a hierarchical modeling and organization of a multiprogramming computer system. In [37, 38] he showed how advantageous it is from a point of view of a software designer to structure a computer operating system as a hierarchy of abstraction levels. These levels are considered as an ordered sequence of abstract machines, each one defined and executed in term of the previous one. A practical example of such a hierarchy of levels of abstractions is reported in [37]. In [40] Dijkstra states that through abstract machines the hardware resources of a computer system can be provided to the upper levels of an operating system in a much more user-convenient way and can therefore help by the construction and validation of large software systems [39, 123].

3.4.1 Design Strategies for Communication Networks

When designing a communication network, one must deal with the inherent complexity of coordinating asynchronous operations communicating in a potentially slow and error-prone environment. It is also essential that distributed systems agree on a protocol or on a set of protocols for determining host names, locating hosts on the network, establishing connections, and so on. We can simplify the design problem (and related implementation) by partitioning the problem into multiple layers. Each layer on one system communicates with the equivalent layer on other systems. Each layer may have its own protocols, or may be a logical segmentation of some protocol. This protocols may be implemented in hardware or software. For example, the three lowest level layers for the logical communications between two computers are often implemented in hardware.

3.4.2 OSI and Internet Reference Model

The traditional way of describing network protocols is to divide a protocol stack into multiple layers. The International Standards Organization (ISO) refers to seven layers with each protocol layer interacting only with the layers below and above with the following descriptions ([36, 148]):

- Application: The application layer is responsible for interacting directly with the users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as

- Presentation: The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions, and half duplex-full duplex modes (character echoing). By using the same machine architectures like in cluster computing this can be prevented.

- Session: The session layer is responsible for implementing sessions, or process-to-process communications protocols. Typically, these protocols are the actual communications for remote logins, and for file and mail transfers.

- Transport: The transport layer is responsible for low-level access to the network and for transfer of messages between the clients, including partitioning messages into packets, maintaining packet order, controlling flow of data and congestion control, and generating physical port addresses. This is often implemented by the TCP protocol.

- Network: The network layer is responsible for providing connections and for routing packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer. This layer is normally implemented with the IP protocol.

- Data-Link: The data-link layer is responsible for handling the frames, or fixed/length parts of packets, including any error detection and recovery that occurred in the physical layer. Ethernet is an example for this layer. with schemas for distributed systems.

- Physical: The physical layer is responsible for handling both the mechanical and electrical details of the physical transmission o a bit stream. At the physical layer, the communicating system must agree on the electrical representation of a binary 0 and 1, so that when data are sent as a stream of electrical signals, the receiver is able to interpret the data properly as binary data. This layer is implemented in the hardware of the networking device.

While the OSI model is a good way of designing and describing network protocols, it is not a very efficient way of implementing them. An implementor would most likely prefer a model where data is transferred reliably directly between applications through an ideal network with infinite bandwidth and zero latency. In practice, neither the complicated OSI model nor the ideal single layer model are feasible.

In a layered approach, the problem of transferring data is divided into multiple smaller problems. This causes difficulties in the implementations, since the individual layers have no knowledge about the overall picture and thus may make bad decisions about how to buffer data. This may cause severely degraded and unpredictable performance [35, 83].

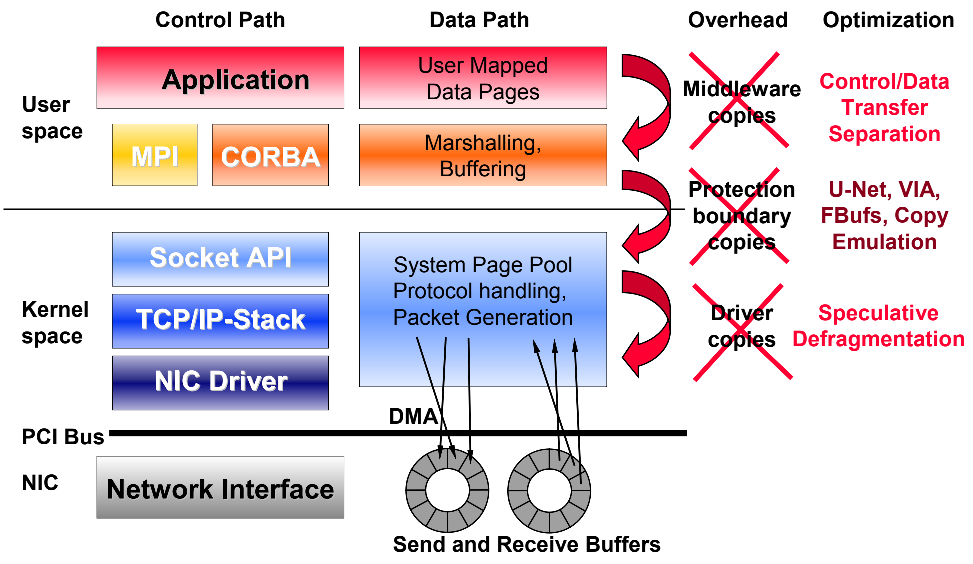


***Figure 3.15**: Control path and data path in a typical system environment.*

Figure 3.15 shows a typical layered communication system where the application communicates its data via a middleware package like MPI or CORBA (see Section 2.4.4 and 2.5.4). The middleware uses a communication service of the operating system like e.g. the Berkeley socket application programming interface to pass the data to the system. The system then adds transport and network protocol headers for TCP/IP and Ethernet and sends the data via the driver to the network interface hardware that implements the data-link and physical layer.

Looking at the data path, we can see the optimizations done to the layering structure on the operating system side. The data still has to be copied from the virtual user memory space into a system page pool. But by omitting the layering in the data path any piece of data has to be touched only once. The main point here is that the system uses the same data structures throughout the whole stack. Therewith also parallelism inside the stack can be maximized.

What still remains is the layering of application and middleware in user space and the networking subsystem of the operating system in the kernel space.

3.4.3 The Middleware Infrastructure

Middleware has emerged as an important architectural component in supporting distributed applications. The role of middleware is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution. It is a connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. Middleware is essential to migrating mainframe applications to client/server applications and to providing for communication across heterogeneous platforms. This technology has evolved during the 1990s to provide for interoperability in support of the move to client/server architectures.

The importance of the topic is reflected in the increasing visibility of standardization activities such as the Object Management Group's Common Object Request Broker Architecture (CORBA) and the CORBA Component Model, the Java Remote Method Invocation (RMI) and Java Beans, Microsoft's DCOM/.NET and the Open Software Foundation's Distributed Computing Environment (DCE).

The main purpose of middleware services is to help solve many application connectivity and interoperability problems. Middleware allows application processes to transparently collaborate across processes and networks despite different system policies, operating systems, programming languages, machine data formats, and networking protocols. In performing this function, it replaces the upper layers five through seven of the OSI model (session, presentation and application layer). As outlined in Figure 3.16 middleware services are sets of distributed software that exist between the application and the operating system and network services on a system node in the network [13]. It provides a more functional set of Application Programming Interfaces (API) than the operating system and network services that allow an application to scale up in capacity without losing functionality.

Distribution transparency is an essential requirement of what a middleware must provide. Specific examples include: Easing the burden of developing distributed applications, typically by providing communication services, correcting endpoint domain inconsistencies, and providing solutions that are reliable and scalable.

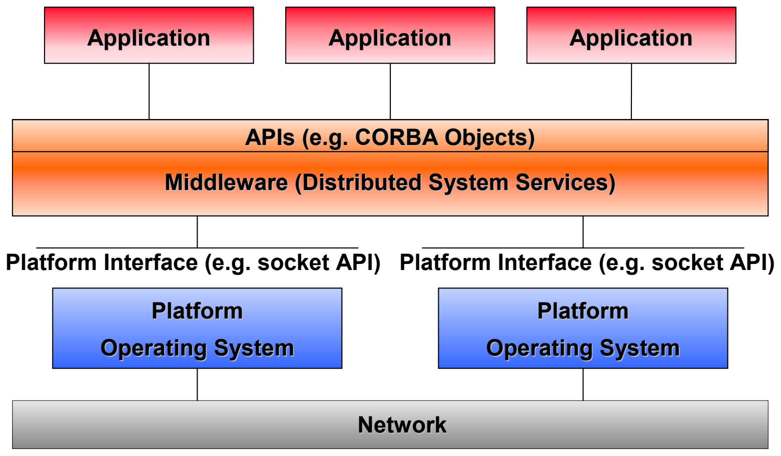

Application
Application
Application
APIs (e.g. CORBA Objects)
Middleware (Distributed System Services)
Platform Interface (e.g. socket API)
Platform Interface (e.g. socket API)
Platform
Operating System
Platform
Operating System
Network

***Figure 3.16**: Use of Middleware*

While there is no formal characterization of middleware, in general there are five categories based on significant standards or products in the marketplace.

- Data Access: Connectivity to databases.
- Remote Procedure Call (RPC): Procedure calls between distributed applications.
- Transaction Processing (TP) Monitors: Distributed transaction management.
- Message Oriented Middleware (MOM): Asynchronous communications between systems.
- Distributed Object Technology (DOT): Object invocations between distributed applications.

Each of these classes of middleware help to reduce the complexity of building certain classes of distributed systems. It is interesting to note that the order also implies the evolution of this technology. Data access and RPC middleware arrived first on the scene, followed by transaction processing monitors and MOM middleware. Distributed object technology is the latest category of middleware, this is the class where CORBA fits. It is based on the best attributes of each of the other categories.

All middleware requires an underlying transparent and associated network protocol as a base on which to build. The data-link and physical layers are typically hidden from the middleware by the transport and network layers. Therefore, they are relatively unimportant to middleware design. But regarding the statement in the last section, that layering can be harmful for efficiency, this is a problem we address in this thesis.

3.5 CORBA and Parallel Computing

Parallel programming systems (APIs and middleware) often use the same distributed hardware platform technology as CORBA: multiple interconnected computers cooperating on an application. Parallel systems have evolved independently from object-based and distributed object computing (e.g. MPI [57]).

This fact prevents parallel applications from taking advantage of the large investment in distributed object software technologies and tools, and also makes it difficult to add parallel parts to distributed CORBA applications. The result of this trend is that parallel programming requires a lot of redundant learning for significantly overlapping technologies (e.g. typed message passing). Similarly, CORBA applications that need parallel subsystems require that the CORBA-programmers learn different tools and technologies for no good technical reason. A unified model for parallel and distributed computing could help to avoid this.

Parallel applications are characterized by a set of processes operating in parallel, usually on parts of a larger data set that is divided up among the participating processes. Data is typically redistributed in a global operation between a set of sending processes and a set of receiving processes, which are sometimes the same single set.

The middleware system of CORBA has generally been considered unsuitable for parallel programming due to its client/server model and due to the lack of peer-to-peer semantics and the resulting difficulty of achieving true concurrency and/or distributed data flow. A number of these issues have been mitigated by recent additions to CORBA such as asynchronous messaging, multithreading, and reactive/recursive ORB implementations (which can process a request while waiting for a reply, all in a single thread). However, not all common interactions needed in high performance distributed computing can be described in today's CORBA model.

Several independent projects have demonstrated CORBA's usefulness for parallel programming ([109, 153]), generally by extending the interactions available and supporting some level of data partitioning [127, 86] (see Section 5.2.2). This has typically resulted in non-standard ORB extensions that are not portable or interoperable. But as the CORBA specification is not static, these project resulted in an new extention and the Data Parallel CORBA specification [113]. We will discuss this in Section 6.2.5.

3.5.1 CORBA versus MPI

In the HPC community the message passing paradigm (with libraries like MPI) is the common starting point for application development. MPI in its basic principle is very simple and it can quite easily be optimized with different well known implementation techniques. Highly optimized MPI implementation exist for most supercomputers and most large scale distributed systems. While MPI is certainly a convenient interface to implement message-passing-based algorithms, a number of problems can be identified:

- It is difficult to integrate external communication into MPI. There is e.g. no straightforward way to integrate the CORBA request processing into the processing of MPI send and receive calls.
- Representing complex data structures is much easier in CORBA. With IDL, complex data structures can be described in a language independent way, and tools generate language bindings to transmit these data automatically.
- Addressing in MPI is on the process level, not on the object level. Thus, in an object-oriented parallel algorithm, addressing of individual objects must be implemented on top of the existing message-passing functionality.

A main problem of MPI is, that it is really just intended to provide a fast message passing facilities but lacks any more sophisticated functionality of a distributed object model. MPI makes applications very difficult to program and is criticized for exposing a low level of machine abstraction that does not map well to any programming model.

These problems seem to be solved in CORBA with most implementations, but there the communication efficiency often remains unsatisfactory. CORBA is primarily used for coupling clients to servers which might be implemented in different languages and run on completely different system architectures and operating systems. CORBA takes the part of negotiating object method invocations and data conversion. As normally many of these remote calls are issued by a client, CORBA ORB optimizations first of all focus on latency as CORBA systems are often latency limited.

Distributed object middleware technology on the other hand provides the largest set of support services. Naming, trading, transaction and security services are all integral parts of CORBA. The best of Data Access, RPC, Transaction Processing Monitor and Message Oriented Middleware have been utilized in CORBA.

3.5.2 Functionality versus Efficiency

In the second part of the thesis we argue for CORBA as an alternative to a message passing interface to give the application developers additional flexibility and the possibility for object oriented component design methods as well as the usage of design patterns. In our application domain we expect the need of large data transfers over high speed networks in cluster. And that is where optimization for high bandwidth with a CORBA system comes in.

As it is not so easily possible to improve MPI and just implement more functionality that eases application development without changing and enhancing the basic specification we chose to take the rich functionality of CORBA and optimize the ORB so that high bandwidth communication is handled efficiently (Figure 3.17).

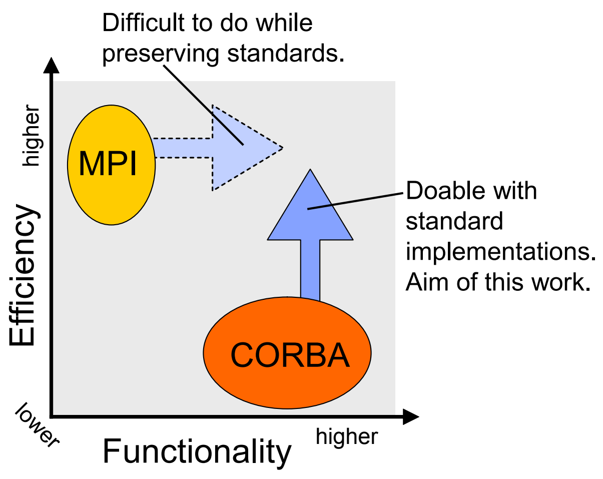

Difficult to do while preserving standards.
MPI
Doable with standard implementations. Aim of this work.
CORBA
Efficiency
higher
lower
Functionality
higher

Figure 3.17: *CORBA versus MPI. While the efficiency of CORBA implementations can be improved, the functionality of MPI is fixed by the specification.*

3.6 Extention of the Zero-Copy Paradigm

In this chapter we suggested that with the right granularity and the right data representation, communication in high speed commodity clusters can be very efficient, in particular when using specialized hardware and optimized communication system software. We further identified the main performance problem with Gigabit Ethernet, namely inefficient transport protocols whose operating system implementations introduce copies in the local memory system. As todays network bandwidths match the memory transfer speeds, this memory copy turned out to be the most expensive bottleneck in high bandwidth communication with Ethernet. The zero-copy paradigm we propose is therefore absolutely needed to achieve efficiency.

Todays complex systems need to be composed out of many components involving different layers that ease the development and divide the applications complexity into concise parts. As a consequence we need a to extend the zero-copy principle to all the involved layers.

Our understanding of a zero-copy architecture really means *zero* data copies through all the involved layers (see Figure 3.18). That means that no copy may occur in the communication system including the hardware drivers and the network and transport protocols as well as by crossing the kernel-user boundary involving the socket interface. In a zero-copy system a middleware that provides an easier to program system abstraction and functionalities to the application may not introduce any copies as well.

Wherever a copy occurs in any layer, this copy is always one copy too much and breaks the whole zero-copy principle. Performance wise the first copy is always the most

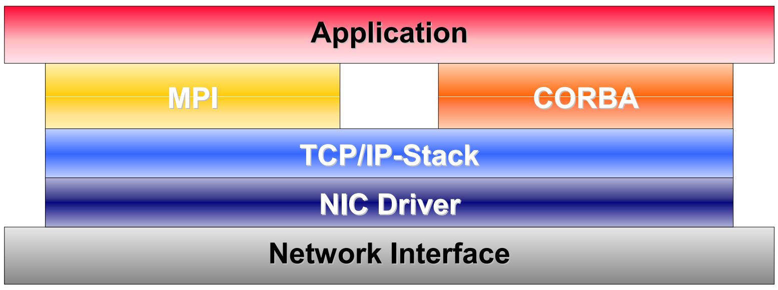


***Figure 3.18**: Layered hardware/software system*

expensive one, considering that data might be readily available in the cache after a first access and render additional copies to be cheaper.

As the system layer and the middleware layer have to be discussed separately and can not be merged into one, we divide the further reflection of the thesis into two main implementation fields for the zero-copy principle: (a) commodity communication hardware and (b) object oriented middleware.

3.6.1 Problems with Commodity Communication Hardware

Section 3.2.3 already discussed the bad performance that can be achieved by using Gigabit Ethernet with a commodity PC compute platform. It becomes more and more important to port traditional applications quickly and easily to parallel infrastructures like clusters by simply substituting a conventional LAN network by the next generation technology. In this mode of operation the performance of a fully standardized protocol stack like TCP/IP is absolutely essential.

The Linux TCP/IP stack and driver architecture already implements many optimizations and the driver even boast itself as a zero-copy architecture. But in its whole the system is a one copy implementation because the system still has to fragment and defragment the user data which is passed to the communication socket. After the fragmentation, the protocol data is generated and sent to the NIC hardware. On the receiver the data is defragmented and copied to the receiver process.

We propose speculation techniques that are well known from processor design to overcome those limitations. These enabling techniques were newer used before in communication software implementations. Chapter 4 discusses how with speculation the nearly hopeless case of a zero-copy TCP/IP stack for simple Gigabit Ethernet hardware becomes feasible.

3.6.2 Problem with Object Oriented Middleware

In Section 3.5 we explained why CORBA is an interesting concept and why this middleware concept should be used in clusters. To prove that it is indeed possible to provide the best possible bandwidth of a communication system through a CORBA layer to an object oriented application design we introduce specialized marshalling plugins in CORBA ORBs which allow the arguments to object messages to be passed by reference through the ORB and further to the system. We achieve this by separating control and data transfer in IIOP.

The granularity of objects that have to be communicated is an important aspect. In Section 3.2 we have stated, that only large contiguous transfers can really benefit from larger bandwidth. This must be considered in the object model of the application. The granularity of the object has to be chosen in a way to permit larger collections or larger grained granularity that is better qualified for high bandwidth communication must be taken into account.

The techniques to overcome the bandwidth limitations of CORBA middleware are subject of Chapter 5. We show that even open source middleware software can meet the requirements realistically. We present techniques to overcome the bandwidth bottleneck of CORBA middleware and introduce a zero-copy ORB implementation.

4

Zero Copy Concepts for Commodity Hardware

As discussed in the previous chapters, Cluster of PCs (CoPs) with Gigabit/s interconnects promise to give workers access to a new game of multimedia applications, but the full potential of Gigabit/s communication technology still remains unused, as long as commodity network adapters like Ethernet NICs or standard protocols like TCP/IP are used. Commodity PCs with their modest memory subsystem performance require either a hardware acceleration or alternatively, a true ”zero-copy” communication software architecture to reach the full network speeds in applications. Correctly defragmenting packets of the various communication protocols in hardware remains an extremely complex task and most widely used protocol stacks for Gigabit Ethernet require at least one last copy for the (de)fragmentation of the transferred network packets preventing a true zero-copy communication system.

This chapter describes a prototype implementation of a defragmenting driver based on the same speculation techniques that are common to improve processor performance with instruction level parallelism. With our speculative implementation we are able to eliminate the last copy of a TCP/IP stack even on simple, existing hardware [91]. We integrated our network interface driver into the Linux TCP/IP protocol stack and added some well known page remapping and fast buffers strategies to reach an overall zero-copy communication architecture. Based on our experience with that driver we can also suggest simple hardware improvements to the network interface to increase the speculation success rates.

An evaluation with measurement data indicates, that we can reduce the CPU load of communication processing for Gigabit Ethernet significantly, that speculation will succeed in most cases and that we can improve the performance for burst transfers by a factor of 1.5–2 over the standard communication software in Linux 2.2.

4.1 Special Challenges of Commodity Hardware

Gigabit Ethernet, like all previous versions of Ethernet, has been designed for an unacknowledged, connection-less delivery service and provides point-to-point connections supporting bi-directional communication including link-level (but no end-to-end) flow control. One of the big advantages of Gigabit Ethernet is the backward compatibility with its predecessor, so called Fast Ethernet (100 MBit/s).

4.1.1 Downside of Backward Compatibility

The downside of the backward compatibility of Gigabit Ethernet to its predecessor is that because of the unreliable service a software protocol layer has to take over the responsibility for reliable connections. This is where TCP/IP comes in and is absolutely needed. A second disadvantage is Ethernets maximum transmission unit (MTU) of 1500 Byte which still remains smaller than the page size of any processor architecture. Ethernet driver systems thus use at least one data copy to separate headers from data, that renders efficient zero-copy techniques useless, unless defragmentation is done in hardware. Therefore the current standards prevent efficient zero-copy techniques from being used and simple copy-techniques are most common in current implementations.

Looking at the resource demands of the error and congestion control mechanisms in Gigabit Ethernet, it might look completely hopeless to implement a fully compatible, fast zero-copy TCP/IP protocol stack with the existing PCI based network interface cards (NIC). Looking more closely at the currently available hardware solutions things do look quite a bit better than expected. Based on link-level flow control, a dedicated network control architecture for high speed communication can be devised so that under certain assumptions, a rigorous true zero-copy protocol might still be feasible with off-the-shelf components.

4.1.2 TCP Transmission Control Protocol

The Transmission Control Protocol (TCP) [79] is the transport protocol from the Internet protocol suite. In this set of protocols, the functions of detecting and recovering lost or corrupted packets, flow control and multiplexing are performed at the transport level. TCP uses sequence numbers, cumulative acknowledgment, windows of packets in transit and software checksums to implement these functions.

4.1.3 IP Internet Protocol

TCP is used on top of a network-level protocol called Internet Protocol (IP) [78]. This protocol, which is a connectionless or datagram packet delivery protocol, deals with host addressing and routing, but the latter function is almost totally the task of the Internet-level packet switch, or gateway. IP also provides the ability for packets to be broken into smaller units (fragmented) on passing into a network with a smaller maximum packet size. The IP layer at the receiving end is responsible for reassembling these fragments. For a general review of TCP and IP, see [141]. Below IP we typically find a driver layer dealing with the specific network technology being used. This may be a very simple layer in the case of a local area network like Gigabit Ethernet, or a rather complex layer for a network such as a token ring. On top of TCP we find a number of application protocols, that make networking available to a user application.

4.2 Related Work in Zero-Copy Communication with Ethernet

A fair amount of previous work in this area resulted from the early projects in network computing. In it many researchers widely acknowledge the observation that badly designed I/O buses and slow memory systems in PCs or workstations are the major limiting factor in achieving sustainable inter-node communication at Gigabit/s speeds [90].

Since today's high speed networks, I/O systems and hierarchical memory systems operate all at comparable bandwidth of about 100 MByte/s, one of the most important challenges for communication system software is therefore to *prevent data copies*.

This section gives a short overview and a classification of existing, but partial zero-copy software techniques as well as the networking technology alternatives which provide optimized hardware support. For a detailed discussion see Chapter 2.

4.2.1 Zero-Copy Software Architectures

Many recent efforts have focused on designing software architectures to overcome copies due to OS protection domains (user mode, kernel mode). The proposed solutions, called *zero-copy*, provide the capability of moving data between application domains and network interfaces without CPU and memory bus intensive copy operations. A variety of approaches to host interface design and supporting software have been proposed. To give an overview of previous and related work we slightly extend a classification in [28].

1. **User-Level Network Interface (U-Net) or Virtual Interface Architecture (VIA)**: Low level hardware abstractions for the network interfaces suggest to leave the implementation of the communication system software to libraries in user space [157, 160, 47, 46].

2. **User/Kernel Shared Memory (FBufs, IO-Lite)**: The technique relies on shared memory semantics between the user and kernel address space and permits to use DMAs for moving data between the shared memory and network interface. Such

drivers can also be built with per-process buffer pools that are pre-mapped in both the user and kernel address spaces [44, 119].

3. **User/Kernel Page Remapping with Copy on Write and Copy Emulation**: These implementations re-map memory pages between user and kernel space by editing the MMU table and perform copies only when needed. They can also benefit from DMA to transfer frames between kernel buffers and the network interface [28, 22, 20].

However the three classes of solutions can not achieve a clean zero-copy on simple commodity hardware. Despite a strict zero-copy handling within the operating system, the implementation techniques mentioned above still fail to remove the last copy in the Gigabit Ethernet driver that is due to packet defragmentation. There are many successful prototypes, but most of them include semantic restrictions for zero-copy buffer management and only work with network technologies supporting large frames. A good overview is given in [21]. A better operating system support for I/O streams is described in [103].

Our work targets zero-copy without semantic restrictions for commodity network interfaces. The basic idea of manipulating the behavior of the Ethernet driver to reduce in-memory copy operations is not new and it has been explored with conventional Ethernet at 10 MBit/s more than a decade ago. These investigations were in the context of:

4. **Blast transfer facilities**: Blasts are special protocols for large transfers [165]. The definition of the protocol requires the recipient to have sufficient buffers available to receive the data before the transfer takes place. It is assumed that the source and the destination machine are more or less matched in speed. The protocol is implemented at the network interrupt level and therefore not slowed down by process scheduling delays. The sufficiently low frequency of the network errors typically allows blasts with full retransmission on error to be acceptable. Acknowledges are sent only after all the fragments of a blast have arrived.

5. **Optimistic Blasts**: In [25], the driver's buffer chain is changed in a way that the headers and the data of the incoming packets are separated by the network interface card. The observation that, during a bulk data transfer over a network, there exists a high probability that the next packet received by the destination host is the next packet in the transfer is correct. This observation allows an optimistic implementation of a bulk transfer in a way, that the network adapter on the destination host is programmed to deposit the data directly to its final destination in the user space.

6. **Transparent Blast Facility**: Although the optimistic blast algorithm is effective, it has serious shortcomings, mainly that it is tailored to work for a single dedicated transport protocol. The algorithm requires that all packets have an identical fixed sized header. Therefore the algorithm is not designed to work in a heterogeneous

network that supports TCP/IP or multiple protocol suites. This lack of transparency was already shown by [105] where all the header data for different protocols which were added to payload were padded to the same lengths. This makes it easy to separate it again and running different protocols. The pages with the contiguous data parts were then re-mapped from kernel to user space. The headers go to kernel buffers like with the Optimistic Blasts.

All these previously described schemes need to take some special action in case the blast transfer is interrupted by some other network packet: The first approach is to limit the blast transfer length and copy the data to the correct location after an interrupted blast.

In the ten years since these investigations on blast transfers, the network and memory bandwidths have increased by two orders of a magnitude and architectural limitations apply. However, the architecture of popular network interfaces for Fast and Gigabit Ethernet has hardly changed. Blast transfers use dedicated protocols and therefore dedicated application programming interfaces. Because this lack of transparency none of the blast techniques have made their way into production clusters or standard operating systems.

7. **Scheduled Transfer Protocol**: A new attempt for a blast transfer protocol is described in Scheduled Transfer Protocol (STP) [77]. STP is a connection-oriented data transfer protocol designed for high-speed local area networks. It was originally designed as part of the ANSI/NCITS standardization work for Gigabyte System Network (GSN), a high-speed network media supporting speeds of up to 6400 MBit/s.

   STP allows to deposit data directly in the memory of the receiver by a clean separation of control and data transfers.

The basic design principle of STP is that as much work as possible should be performed by the transmitter, and the receiver only needs to verify the incoming packet and place the data into the correct buffer. Before any data transfer happens, small control messages are transmitted to pre-allocate buffers at the receiver before the data movement begins. The data can then be directly moved from the network into host memory. This reduces the workload of the receiver considerably and makes hardware acceleration relatively simple to implement as the state information for the protocol is very low. This also makes zero-copy receives possible. To accomplish these goals, STP must make several compromises. The network is assumed to be secure, reliable and have very low latency.

STP can be used through several different APIs, including BSD sockets and libST which is a OS bypass library designed for use with STP. The protocol is also being adopted as a solution for networked storage (SCSI over STP (SST) standard). An implementation of STP over Gigabit Ethernet for Linux [125] is currently available as a kernel patch but only works with programmable Ethernet interfaces which are rarely standard commodity equipment.

Our work is different, since it aims at performance improvements by improving the integration of fast transfers with existing hardware and existing protocol stacks and, as a novelty, we add the distinct viewpoint of speculative protocol processing with cleanup code upon failure.

This clearly leads to a better understanding of the hardware support. We therefore will point out two simple and effective enhancements of the current network adapters which would keep the simplicity of the hardware but improve the ability so that efficient zero-copy communication for special blast over TCP/IP become possible.

4.2.2 Gigabit Networking Alternatives and their Solution for Zero-Copy

For System Area Networks (SAN) of the clusters of PC platform a few highly attractive SAN alternatives to Gigabit Ethernet are available or at least proposed an in the state of being developed.

1. **SAN Alternatives to Gigabit Ethernet**: Those alternatives comprise Myrinet [16], SCI [41], Giganet [75], ATM-OC12 and InfiniBand [82]. For a more detailed overview see Chapter 2. There are some relevant architectural differences between these interconnect technologies and Gigabit Ethernet. In most special purpose SAN technologies the transfers between the host and the application program can be done in blocks whose size is equal or larger than a memory page size; this is fundamental for zero-copy strategies and efficient driver software.

Myrinet interconnects support long packets (unlimited MTU), link level error and end-to-end flow control that is properly handled by deadlock free routing in the wormhole switches. Furthermore, the Myrinet network interface card provides significant processing power through a user programmable RISC core with large staging memory. Under these circumstances there is much less justification for a TCP/IP protocol stack. Similarly the bulk data transfer capability of SCI interconnects relies on hardware for error and flow control in the network to avoid the problem of fragmenting packets. Giganet also incorporates a supercomputer-like reliable interconnect and pushes a hardware implementation of VIA to provide zero-copy communication between applications from user space to user space. In contrast to these supercomputing technologies, the ATM-OC12 hardware operates with packet losses in the switches and highly fragmented packets (MTU of 53 Byte). However, with such fast links and such small packets, there is no hope for defragmentation in software and ATM adapters must provide this functionality entirely in hardware. The rich ATM functionality comes at a hardware cost, that might be too high for most PC clusters. ATM technology disappeared almost entirely from high performance computing but remains the premier option for wide area network backbones.

InfiniBand is a new SAN technology sponsored by nearly all major computing vendors, including Compaq, Dell, Intel, IBM, Microsoft, Sun and HP. It supports a wide range of applications from a switched backplane interconnect for a single host (replacing the PCI bus) to connecting a large number of independent hosts and I/O components (replacing Ethernet and Fiber Channel). The address space for different nodes on a InfiniBand network is done using IPv6 addresses and therefore an integration of WAN and SAN with simple bridges is possible. The first implementations of InfiniBand are scheduled to arrive in 2002.

InfiniBand is not just a hardware solution. The standard includes a full featured protocol layer that is very similar to software-based solutions. It uses basically the VIA primitives for its operation at the transport layer.

InfiniBand comes in three different versions, 1x, 4x and 12x, which run at 2.5 GBit/s, 10 GBit/s and 30 GBit/s, respectively. On the physical level it uses 4096 Byte frames, but supports reliable transfers in units of up to 2 GBytes.

2. **Silicon TCP/IP**: A similarly expensive hardware solution is SiliconTCP^{TM} [80]. It implements a TCP/IP stack in hardware which leads to the benefit of very low main processor utilization but in current implementations it cannot keep up with a Gigabit/s rate of fast interconnects. For most protocols including TCP/IP, however, implementing hardware acceleration is challenging, defining an appropriate layer for dividing the software and hardware components is very difficult. That's why moving the entire protocol into hardware is an certainly option that would make it easier to achieve full performance. But for economic (and often technical) reasons the option is rarely exercised.

3. **Hardware Accelerated TCP/IP**: Despite the difficulties involved, hardware accelerated TCP/IP is still very attractive. This line of designs does not intend to implement a full TCP/IP stack in hardware, but rather offloads the fast path of protocol processing.

   The simplest form of acceleration, the hardware checksumming mechanism, is now a standard feature in every new NIC, although the feature is often unusable due to hardware problems. For the majority of client-server applications, such as HTTP or NFS, where a server is connected to a high-speed network an just serves files to multiple clients on slower networks, those hardware checksums can enable zero-copy transmits.

Although the company built on SiliconTCP was a failure, the route of TCP/IP implemented entirely in hardware will undoubtedly be tried again in the future despite past experience that has shown it to be a bad idea due to technical reasons, e.g. entire systems in a chip. Many protocol bugs are only apparent on fast networks and remain impossible to fix for anyone except the vendor. Upgrading a system to new protocols such as IPv6 remains impossible without upgrading the hardware. The extra messaging required between the OS and the NIC to guarantee a proper flow control also often removes any benefit gained. But despite all those difficulties involved, hardware accelerated TCP/IP is still very attractive and the approach starts to look promising again facing all the cheap network processors in development right now. Furthermore FPGAs become large and cheap enough to accommodate a protocol processing engine. Several hardware manufacturers already announced chips that implement some support for TCP in hardware. But it will still take time until this technology becomes available and will be built into commodity PCs in a second step. So there is plenty of opportunity for a new software solution to be successful.

Instead of implementing the full TCP/IP functionality in hardware simpler lightweight protocols were developed as already mentioned in the last section.

4. **Ethernet Lightweight Protocols with Hardware Support**: Lightweight protocols count on faster in-time packet delivery with lower error rates and hardware flow control enabled as guaranteed by newer interconnect technologies. That does not only simplify the implementation of such protocols and minimizes the footprint of such stacks but also allows to run these stack engines on slow and simple processors as available on some current Ethernet adapters. Ethernet Message Passing (EMP) [137], Arsenic [126] or STP on Ethernet [125] are examples of such protocols that rely on adapter hardware support.

For a while Alteon Web Systems, now owned by Nortel Networks, produced a Gigabit Ethernet network interface chipset based on a general purpose embedded microprocessor design which they called the Tigon2. It is novel because most Ethernet chipsets were mostly fixed full custom hardware designs, using a standard descriptor-based host communication protocol. A fully programmable microprocessor allows for much flexibility in the design of a communication system. This chipset was sold on boards by Alteon, and was even used in motherboards or NIC designs by other companies, including Netgear.

The only thing that is mandatory for the usability of such adapters in research is that the hardware implementor provides the technical specifications of the processors and tools to implement and change the protocol firmware. This was the case with Alteon for the first two generations allowing research and development in that area without producing ones own hardware. Many research groups provided implementations and optimized versions of their protocol on this hardware. Even the Jumbo Frame idea (see next paragraph) was first implemented with this adapter. But nevertheless the latest update of the chipset is not Open Source anymore while the old hardware is discontinued. Unfortunately the entire software architecture depends on the vendor and the Alteon NIC was taken off the market. This means for many researchers to continue with old hardware or implement their own design anyway.

Again a slightly different approach is to take a special purpose high speed networking adapter like Myrinet and program it the way that it behaves like an Ethernet adapter and can send TCP/IP packets.

5. **Zero-Copy TCP with Hardware Support**: The Trapeze[26, 61] project shows experiences with high-speed TCP/IP networking on a gigabit-per-second Myrinet network, using a Myrinet messaging system called Trapeze. It explores the effects of common optimizations above and below the TCP/IP protocol stack, including zero-copy sockets, large packets with scatter/gather I/O, checksum offloading, message pipelining, and interrupt suppression.

The Trapeze project is said to hold the world record for ”the first demonstration on public record of end-to-end TCP/IP at faster than a gigabit-per-second”. The 1.147 Gigabit/s TCP/IP data rates in the netperf benchmark was performed over a Myrinet LAN using Myricom's 64 bit Myrinet/PCI interfaces in Compaq XP-1000 (Alpha 21264) workstations running FreeBSD. While native Myrinet protocols can sustain Gigabit/s data rates these are much more difficult to achieve when using the standard TCP/IP protocols. The Duke software and the Myrinet control program employ zero-copy, gather-scatter, and checksum-offload techniques, that are all assisted by the protocol engines and hardware features of the 64-bit Myrinet/PCI interfaces. Myrinet interfaces are an added value costing $1000 per cluster node and can hardly be labeled commodity hardware.

4.2.3 Zero-Copy Extentions to Existing Technologies

To overcome the problem of packets that are smaller than a memory page, some Gigabit Ethernet vendors propose a change of the standard by simply introducing larger frame sizes.

1. **Jumbo Frames**: Jumbo Frames [4] by Alteon Websystems is an attempt to establish a new Ethernet standard with an increased frame size. In this solution the maximal Ethernet packet size (MTU) is increased to 9000 Byte and a proprietary network infrastructure supporting them is required. In addition to reducing the number of interrupts, Jumbo Frames are large enough to accommodate for entire memory pages.

This technique is primarily useful for applications that have their focus on sending data like web servers, while the receive part does not profit much. Jumbo Frames make the software defragmentation into whole memory page sizes needless but nevertheless, the concept of Jumbo Frames does not solve the problem of header/payload separation hereby contradicting the idea of using mainstream networking technologies in a cluster of PCs.

As the Jumbo Frame extention is just specified for Gigabit Ethernet, IP routers or switches have to overtake the task of fragmenting the large frames into smaller packets when forwarded to a Fast Ethernet link. This functionality may exist in the products of a few vendors but it is often not available in current switching hardware which limits the hardware choices on the market to a minimum. Furthermore, many high-end SAN interconnects used wormhole forwarding in the past, where most LAN Ethernet switches are built with simple store-and-forward packet handling. In packet switched networks, the presence of Jumbo Frames will result in high latency not only for the Jumbo Frames itself but also for all small packets traveling in the same network.

Instead of demanding ever increasing packet sizes in high performance networking we suggest to incorporate a modest additional hardware support for defragmentation of small Ethernet packets (similar to ATM cells) which results in lower overall latencies and high bandwidth transfers for highly fragmented packets. As such interfaces are still not available today, we enlist techniques of speculation to go the route of efficient defragmentation at the driver level in software as this is already properly defined by the IP over Ethernet standard. Due to speculation this can be done with the existing network interface hardware.

As a last point I shall mention that the Transmission Control Protocol is improving as well. An aspect of high-speed networking, which will not be dealt with in this thesis, is performance on wide-area networks such as the Internet.

2. **Improvements to TCP**: The view of high speed network performance over the Internet changes the problem from being able to send packets as fast as possible from user buffer to user buffer into being able to accurately estimate the maximum rate data can be transferred without packets being dropped and recovering from any errors as quickly as possible. Current research in this area includes improvements to the TCP selective-acknowledgment option [55] and congestion management [56], choosing the correct size for socket buffers and the use of parallel streams as well as developments for High Speed TCP like larger Congestion Windows or limited Slow-Start.

TCP is able to accomplish this with reasonable success, although some improvements are still needed. The problem is, that all modifications made to it must be fully backwards-compatible with existing TCP implementations, work even on slow networks and maintain the politeness of TCP (which prevents the Internet from collapsing). These restrictions make any radical changes to the protocol impossible.

4.3 Enabling Zero-Copy for Commodity Ethernet Hardware

In order to achieve networking performance between 75 and 100 MByte/s with a standard Gigabit/s network interface, a zero-copy protocol architecture is absolutely necessary. With the common restriction of the MTU to 1500 Byte (which is less than a page size) and the usual hardware support of a descriptor based Ethernet network interface, it seems to remain impossible to solve the problem of a true zero-copy transfer because of simplistic DMA hardware cannot even reliably separate protocol header and payload data.

It may be possible to send data without local memory copies which is done with current expensive server hardware. This is especially useful for Web and file servers which have to ship loads of data to many clients but just receive small request packets from these clients. This is different in our scenario of high performance distributed computing where one single connection from a producer to a consumer requires the full available bandwidth. In this scenario the receive operation has to be efficient as well and therefore optimized for zero-copy similarly to the send operation. Therefore our new method in this thesis must address zero-copy in the sender and the receiver as well.

4.3.1 Speculative Processing for Packet Defragmentation

To overcome the restrictions given by standard network technologies and the simple hardware functionalities, we propose to use *speculative processing* in the receiver driver to defragment IP-fragments with the current hardware support. In our implementation of IP over Ethernet the driver pretends to support an MTU of an entire memory page (4 KByte) and handles the fragmentation into conventional Ethernet packets at the lowest possible level. By speculation we assume that during a high performance transfer all Ethernet packets will arrive free of errors and also arrive in the correct order so that the receiving driver can put the payload of all fragments directly into the final destination. Once the defragmentation problem is solved and the packet is properly reassembled in a memory page, all well known zero-copy techniques can be used to pass the payload further to the application.

As with all speculative methods, the aim of speculative defragmentation is to make the best case extremely fast at the price of a potentially more expensive cleanup operation if something went wrong. We will show with application statistics that the best case is indeed the common case. If either the data is smaller than a page or a speculative zero-copy defragmentation does not succeed because of interfering packets, the data is passed to a regular protocol stack to be handled in a conventional one-copy manner. With the current hardware only one high performance stream can be processed at a time with optimal performance. The other streams are throttled to lower performance. On other network technologies like e.g. Myrinet, this is also the case since large transfers are non interruptible. The zero-copy transfers must be scheduled explicitly or implicitly between two nodes (an implicit solution is given in Section 4.5). A main point of this implementation is that it can always coexist with normal TCP/IP and ARP networking traffic while running without a performance penalty.

4.3.2 Gigabit Ethernet and its NICs

In our experimental cluster of PCs we use off-the-shelf 400 MHz Pentium II PCs, running Linux 2.2, connected via Gigabit Ethernet by fiber optic cables. Our Gigabit Ethernet test bed comprises a SmartSwitch 8600 manufactured by Cabletron and GNIC-II Gigabit Ethernet interface cards manufactured by PacketEngines.

Gigabit Ethernet is successful for several reasons. The technology is simple, which translates to high reliability and low maintenance cost as well as a reasonable cost of entry (e.g. compared to ATM). Gigabit Ethernet is layered on top of the already developed and tested physical layer of enhanced ANSI standard Fiber-Channel optical components that are well proven for connecting a network host-adapter to a central high performance packet switching backplane.

The PacketEngines NICs use the Hamachi Ethernet interface chipset that is a typical Gigabit Ethernet controller. Besides some buffering FIFOs towards the link side, the controller chip hosts two independent descriptor-based DMA processors (TX and RX) for streaming data to and from host memory without host intervention hereby reducing the CPU load necessary for network handling. Advanced interrupt coalescing techniques reduce the number of host interrupts to a minimum and multiple packets can be handled with one single interrupt. The controller chip also detects TCP/IP protocol frames and correctly calculates the necessary checksums while forwarding the packets to the host. Some large internal FIFOs and an extended buffering capability in external SRAM chips maximize the autonomy of operation and limit the chances of packet loss when the host processor is heavily loaded.

4.3.3 An Implementation of a Zero-Copy TCP/IP Stack

For a prototype implementation of our zero-copy TCP/IP stack with driver level fragmentation we use several well-known techniques as indicated in Section 4.2 and Chapter 2 — in particular “page remapping” [28] and copy emulation [22] or as an alternative the “fast buffer” concept [44] enhanced by container shipping ideas [7]. The two different zero-copy techniques demonstrate that the speculative defragmentation technique is an independent achievement and that it works with different OS embeddings.

4.3.4 Changes to the Linux TCP/IP Stack for Zero-Copy

The Linux 2.2 stack is similar to the BSD implementation and already applies a single-copy buffering strategy which required only a few changes to make it suitable for zero-copy. We added a new socket option allowing applications to choose between the new zero-copy sockets and the traditional ones.

On the sender side, the data to be sent is copied and checksummed out of the application buffer into a so called *socket buffer* in kernel space. Instead of copying the data, we remap the whole page from user to kernel memory and mark it copy-on-write to preserve the correct API semantics. A pointer (zc_data) to the remapped page is added to the socket buffer data structure to keep our changes transparent to the rest of the stack (Figure 4.1). The original packet data field is still used by the stack for the generation of the headers.

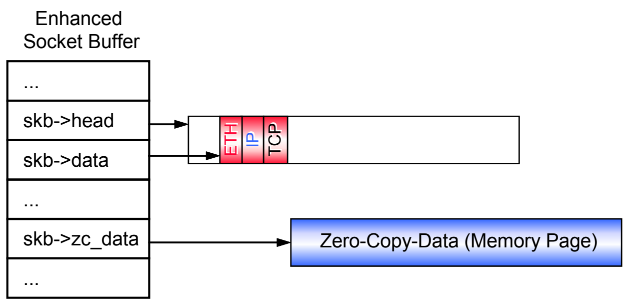


***Figure 4.1**: Enhanced socket buffer with pointer to a zero-copy data part (memory page)*

We do not checksum data as this can be offloaded to the hardware. This can be activated by just setting a flag in the socket buffer.

On the receiver side, the original device driver controls the DMA operation of the incoming frame to a previously reserved socket buffer during the interrupt. Later, after the time-critical interrupt, the data is immediately defragmented in the IP-layer and passed to the TCP-layer. When the application reads the data from the socket, it is finally copied and defragmented to the application buffer. This is different in our implementation where we reserve an entire memory page from the kernel to store a series of incoming fragments. Once this page holds an entire 4 KByte IP packet, it is passed through the IP and TCP stack where it normally remains unchanged and the remapped to the address space of the receiving application. Remapping instead of copying is possible once the following conditions are met:

1. The user buffers are page aligned and occupy an integral number of MMU pages.
2. The received messages must be large enough to cover a whole page of data.
3. The `zc_data`-pointer must point to the data of the zero-copy socket buffer.

If one of the conditions is violated our implementation falls back to the normal operation of the Linux TCP/IP stack and preserves the copy semantics of the traditional socket interface.

4.3.5 Speculative Defragmentation in Hardware

Our fragmenting Ethernet driver manages to send and receive an entire memory page and further features header separation in hardware. The TCP protocol stack therefore automatically generates zero-copy packets of 4 KByte whenever possible and the driver decomposes them into three IP-fragments, each fragment using two DMA descriptor entries — one for the header data and one for the application data. Therefore six descriptors are used to transmit or receive one fragmented zero-copy packet. This scheme of descriptor management permits to use the DMA-engine of the NIC in order to fragment and defragment frames directly from and into memory pages that are suitable for mapping between user and kernel space.

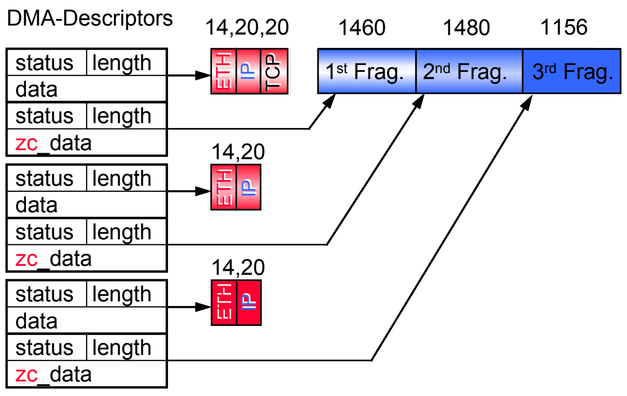


***Figure 4.2**: Descriptor list with 6 entries showing a defragmented 4 KByte packet. The header data consists of an Ethernet and an IP part, in the first packet additionally a TCP part. The numbers indicate the length of the parts in Byte.*

A descriptor entry comprises fields to a pointer to the data in the buffer, for status and for the buffer length. Figure 4.2 shows a snapshot of a descriptor list after the buffers were written by the DMA. With an *End_Of_Packet* flag in the descriptor status field the controller allows for the bytes of an Ethernet frame to extend across several buffers. Thanks to this indicator it becomes possible to automatically separate the headers from the payload.

4.3.6 Subject of Speculation

There are two points where speculation is needed in our implementation. The problem is that descriptors must be statically pre-loaded in advance without knowing what the next packets to arrive will contain. With that a proper separation of header and data is only possible by guessing the length of the header[1]. That means we first speculate on the precise packet format (i.e. the protocol, the header-lengths and the data-field sizes). For the zero-copy implementation the length of the Ethernet-, IP- and TCP-header fields must be pre-negotiated and assumed correctly.

When choosing another protocol (e.g. IPv6 or a special purpose protocols) the only thing that has to be altered in the driver are the length values of the header and data fields that are speculated on.

[1]The Hamachi chip does protocol interpretation at runtime to calculate checksums, but we found no way to use this information to control the DMA behavior. We hope that future Gigabit interface designs have better support for this.

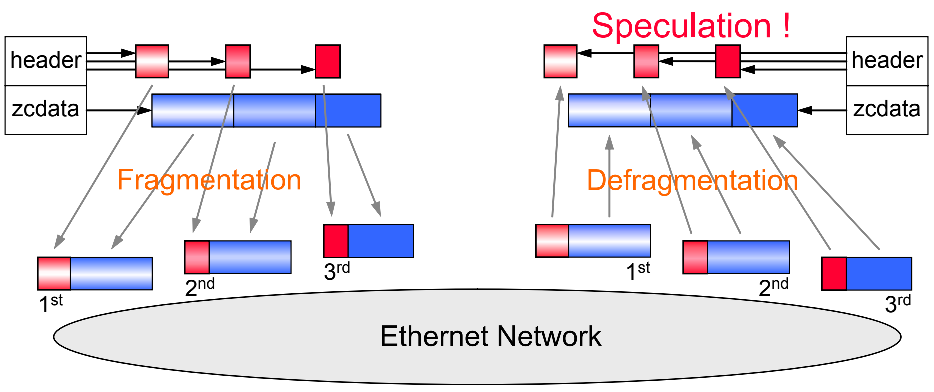


***Figure 4.3**: Fragmentation/Defragmentation of a 4 KByte memory page is done by DMA through the interface hardware.*

The second point we are speculating about is that that all fragments of a whole page arrive in proper order. As described above we engage the well specified technique of IP-fragmentation. This means, that each 4 KByte packet will be fragmented on the lowest driver level to 3 IP-fragments and sent over the network. As discussed in [88] fragmentation can be harmful to the performance for long distance traffic where packet loss occurs or different routes are parallelly used and packet reordering is needed. In our environment of a cluster network with a central switch this is not the case at all as packets errors are very seldom (fiber connectors), packet loss is absolutely minimized by link level flow control by the Ethernet hardware and reordering does not occur. The only case where the speculation could be wring is when interfering packets occur (e.g. ARP packets or more than one stream to the same receiver).

In case of a mis-speculation a software cleanup is performed. The scheme has the nice property that data is never lost. That means by introducing the copy which was first optimized away the packets can be cleanly reordered again.

If an incoming frame does not match the expected format of an IP fragment or contains unexpected protocol options, the zero-copy mode is aborted and the packet is passed to the regular protocol stack and copied. In the current implementation every unexpected non-burst packet causes zero-copy operation to be disrupted.

4.3.7 Packet Transmission and Reception

The responsibility of fragmenting packets of 4 KByte payload into appropriate Ethernet fragments is taken away from the standard IP-stack and given to our NIC driver. The driver logically emulates a virtual network interface that allows to send larger Ethernet frames. The fragmentation is done in a standard way as specified by the IP protocol by setting the *More_Fragments* flag and the proper offsets in the IP-header within the original

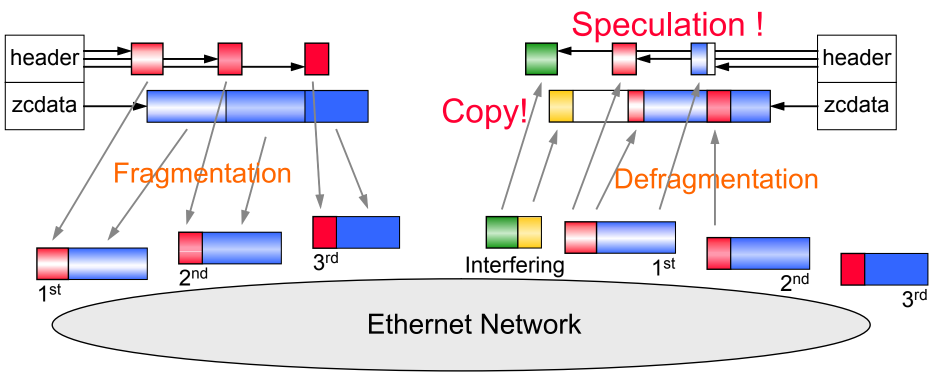


***Figure 4.4**: When interleaving packets occur, some data ends up in a displaced location because of a mis-speculation. The driver therefore introduces the copy that was optimized away again to sort things out.*

packet as outlined in Figure 4.3. The status of transfer (fast or regular) and packet length is known to the sender, so there is no speculation involved here. That means that moving the fragmentation/defragmentation from the IP-stack to the device driver allows to offload the time consuming task to the NIC hardware.

Upon packet arrival, the controller logic transfers the header and the payload into the buffers in host memory designated by the speculatively pre-loaded receive descriptor list. After all the fragments are received or, alternatively, after a timeout is reached, an interrupt is triggered and the driver checks whether the payloads of all received fragments have been correctly written to the corresponding buffer space. Those checks involve the protocol family, the IP-packet IDs and the fragment offsets.

In the success case, if all the fragments have been received correctly, the IP-header is adapted to the new 4 KByte frame and the packet is passed further to the IP-stack. In the failure case, if the driver has received an interfering packet between the individual fragments of a 4 KByte frame, some data ends up in a displaced location because of a mis-speculation (see Figure 4.4). As soon as the device driver detects this condition the receiving DMA is stopped and the data has to be copied out of the wrongly assumed final location into a new socket buffer and afterwards passed forward for further processing. Hence back pressure is applied to the link and a packet loss can be prevented in most cases due to buffers in the NIC and the Gigabit Switch.

4.3.8 Interrupt Coalescing and Adaptive Latency Optimization

A downside of the speculative defragmentation as proposed in this section is depicted in Figure 4.2. As the hardware cannot separate header and payload on its own, we need two descriptors instead of one to store one packet on the host. But the descriptor is not the problem itself as descriptor lists have a very small footprint and they can be reused as far as the data is consumed by the application or the TCP stack in case of header information. But what takes more time is the overhead initiated by each DMA transfer because the PCI bus has to be allocated and arbitrated for each transfer. As we do not want to use Jumbo Frames we have to use 6 DMA descriptors to store one memory page to the host which introduces the double amount of receive interrupts.

The hardware designers were aware of this bottleneck for Gigabit/s networks with small MTUs and have incorporated hardware support for interrupt coalescing into their adapters. The Hamachi chipset uses a very advanced coalescing engine which allows to issue interrupts after the reception of a specified number of packets or alternatively after a certain time has passed.

The combination of these two values allow to gain control over the number of interrupts generated by the adapter. As while blasting for a zero-copy transfer it does not make sense to issue an interrupt for less than an entire 4 KByte packet, which means more than 6 DMAs transfers, we set the coalescing in multiples of 6. Of course this does not make sense for latency sensitive communication phases like barriers or phases where less traffic is on the net. There the latency is wasted and bounded only by the timeout time set to the coalescing engine when less than 6 packets arrive. We therefore devised an adaptive scheme of setting the coalescing values. The driver inherently sets the value to a small number permitting best latency when less traffic is handled over the adapter. This allows to profit from very good latencies while still using the coalescing technique to reduce interrupts during high bandwidth blast transfers.

4.4 Performance Results

4.4.1 Performance Limitation in PCI based PCs

The two Gigabit/s networking technologies introduced earlier are able to provide at least a Gigabit/s transfer rate over the physical network wires. Therefore — in theory — a 1000BaseSX Ethernet allows transfer rates of about 125 MByte/s less protocol overhead.

The external PCI bus in current commodity PCs runs at 33 MHz with a 32 bit data path permitting data transfer rates of up to 132 MByte/s in theory. The maximal performance in practice can reach 126 MByte/s, as we measured for large burst transfers between a PCI card and the main memory of the PC.

The performance for a memory to memory copy by a Pentium II CPU is at 92 MByte/s for the Intel 440 BX chipset operating an SDRAM based memory system clocked with 100 MHz. Therefore we can predict a theoretical performance limit of 100 MByte/s for a zero-copy implementation, 50 MByte/s for a one-copy implementation and 33 MByte/s for a two-copy protocol stack.

As workstation main memory bandwidth has been and is expected to continue increasing more slowly than the point-to-point bandwidth of communication networks, copying between system and application buffers is and will be the major bottleneck in end-to-end communication over high-speed networks.

4.4.2 Measured Best Case Performance (Gains of Speculation)

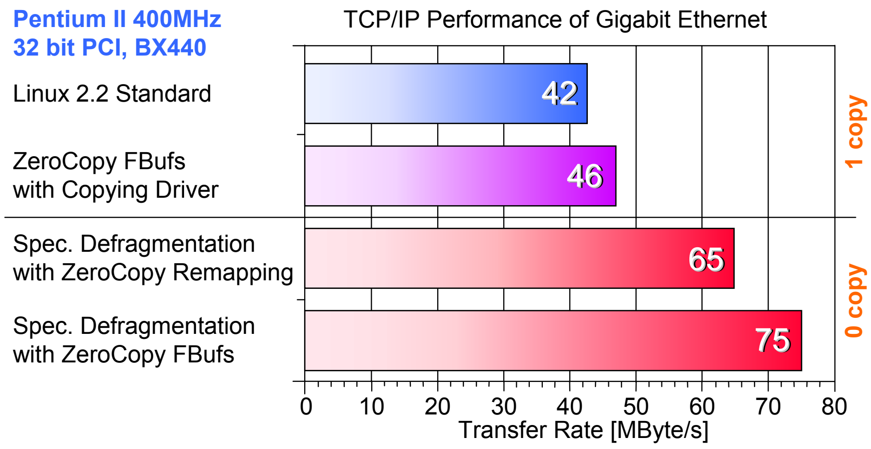


***Figure 4.5**: Throughput of large data transfers over Gigabit Ethernet. In combination with a zero-copy interface (FBufs) the throughput is increased from 42 MByte/s to 75 MByte/s. If the zero-copy embedding as well as the speculative defragmentation is used alone the performance does not increase much, since data is still copied in the driver for packet defragmentation or in the upper layer of the system between user and kernel space. Only the combination of the two techniques enables the higher speeds of true zero-copy communication from 35% to 60% of a true Gigabit/s. The figures characterize the performance on two 400 MHz Pentium II PCs with an Intel 440 BX chipset and a 32 bit/33 MHz PCI-bus.*

Regular distributed applications executing on top of the standard Linux 2.2 kernel achieve a transfer rate of about 42 MByte/s for large transfers across a Gigabit Ethernet (see Figure 4.5). The effect of the performance increase has two tightly linked factors: (a) use of the zero-copy OS techniques and (b) use of the speculative defragmentation. With an OS support for zero-copy alone (e.g. remapping with copy on write (COW)) the performance is still disappointingly low and after all did not improve much, as with Ethernet there is still the defragmenting copy. About the same performance is achieved with the speculative defragmentation alone (without a zero-copy embedding into the OS). In both cases the bandwidth increases to 46 MByte/s as our current implementation of the speculative defragmenter attempts to receive three packets in a row and therefore reduces the interrupt load on the receiver by delaying the generation of an interrupt until at least three packets have been received or a timeout is reached. The limit of three packets is considerably lower than the very large transfers considered in previous work about blast protocols. This improves the minimal length for half of peak speed (half-length) considerably but still fails to deliver good bandwidth for the asymptotical case of large transfers.

To achieve the performance leap we have to combine both technologies to eliminate the last copy. After integration of our defragmenting driver into the zero-copy OS environment, the performance of the TCP/IP stack is increased to 65 MByte/s for fast transfers in a “page remapping” environment and 75 MByte/s in a “fast buffers” environment. The “page remapping” approach is slightly slower than the “fast buffers” approach since expensive memory mapping operations are performed during the transfer in the first case and during startup in the second case.

4.4.3 Performance of Fallback (Penalties when Speculation Fails)

A first “backward compatibility” fallback scenario measures the performance of a sender that dumps fragmented 4 KByte TCP packets to an unprepared standard Linux receiver. As mentioned before, we use standardized IP-fragmentation and so every receiving protocol stack is able to handle such a stream without any problems at normal speed (see Figure 4.6). Actually the performance of this fallback scenario is higher than the standard protocol stack; the negligible improvement is probably removing a bottleneck in the standard sender fragmenting in the driver rather than in the TCP/IP protocol stack.

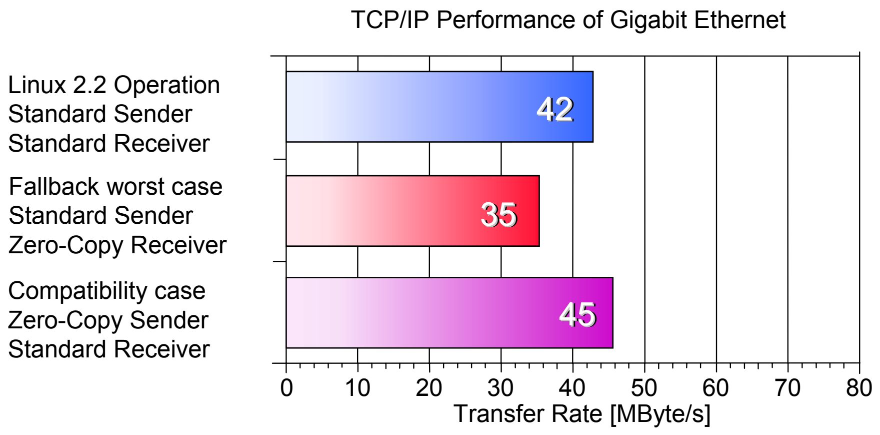


***Figure 4.6**: TCP throughput across a Gigabit Ethernet for two fallback scenarios. There is no penalty for handling sender fragmented packets at an unprepared receiver in normal receive mode. Only if a standard sender is interrupting a burst into a zero-copy receiver, cleanup and resynchronization after each packet is needed. This case should be infrequent and remains unoptimized, but it still performs at 35 MByte/s which is not much slower than the standard implementation at 42 MByte/s.*

The more interesting fallback case is when speculation in the driver fails entirely and a fallback into the cleanup code at the receiver is required. The maximum cost of this worst case is depicted with a standard sender transmitting to a receiver in zero-copy mode. The overhead of the cleanup code reduces the performance from 42 MByte/s to about 35 MByte/s. If the device driver detects that a packet not belonging to the current stream was received as zero-copy packet, two actions need to be taken: (1) The reception of more packets must be stopped, the descriptor lists reinitialized and the reception of packets restarted, (2) the scattered fragments must be copied from the zero-copy buffers and passed on to the regular IP-stack. This explains the small performance reduction.

Packets interfered	Failed ZC-Packets	Bandwidth [MB/s]
0	2	75
100	15	73
10000	28	63

***Table 4.1**: Effect of interferences on a transfer with the* `ttcp` *benchmark sending 100’000 zero-copy packets containing 4096 Byte. The bandwidth is still much better even if the zero-copy transfer is highly interrupted.*

For well speculated or pre-scheduled communication, the cleanup scenario is already highly unlikely. By an appropriate network control architecture that coordinates the senders and receivers in a cluster (see Section 4.5) the probability of such occurrences can be reduced substantially. Table 4.1 shows the bandwidths achieved with infrequently interrupted zero-copy transfers. As depicted in the Table small the transfer bandwidth even if interfered is still much better. This proves that different transfers and concurrent protocols like ARP or ICMP can be handled without much influence on the fast zero-copy transfer.

With a simple packet filter through a match CAM register in the NICs (as proposed and described in Section 4.6) which would separate any burst streams from all the other traffic the miss rate could be further reduced and the average achieved bandwidth would more or less match the maximal achievable bandwidth.

4.4.4 Rates of Success in Real Applications

Table 4.2 shows the packet-arrival traces of two applications running on our cluster. The first trace is taken from a distributed Oracle database executing query 7 of a TPC-D workload (distributed across multiple nodes of cluster by a SQL parallelizer) and the second trace is the execution of an OpenMP SOR code using the TreadMarks DSM system (distributed shared memory). Both applications have been traced for their communication patterns that show different results.

Application trace		Oracle running TPC-D			TreadMarks running SOR		
		Master	Host 1	Host 2	Master	Host 1	Host 2
Ethernet	total	129835	67524	62311	68182	51095	50731
frames	large (data)	90725	45877	44848	44004	30707	30419
	small (control)	39110	21647	17463	24178	20388	20312
Without Network Architecture							
Zero-copy	potential	26505	12611	13894	14670	10231	10135
packets	successful	12745	12611	13894	14458	10225	10133
	success rate	**48%**	100%	100%	99%	>99%	>99%
With Network Architecture							
Zero-copy	potential	26505	12611	13894	14670	10231	10135
packets	successful	26028	12591	13894	14458	10225	10133
	success rate	98%	>99%	100%	99%	>99%	>99%

***Table 4.2**: Study about the rate of success for speculative transfers based on application traces (numbers signify received frames/packets) without and with a driver level control architecture. The TreadMarks application prevents interferences of fast transfers whereas the TPC-D benchmark needs a control architecture to guarantee a predication rate that makes speculation worthwhile.*

In the Oracle case with TPC-D workload, two bursts containing results from queries are simultaneously communicated back to the master at the end of the distributed queries. This leads to many interferences (see Figure 4.7), which need to be separated by a network control architecture. In the next section discusses such a control architecture which prevents stream interferences. The result with this control architecture shows that the mis-speculations mostly disappear which proves that the scheduling on driver level works fine.

In the TreadMarks example, entire pages are distributed over the network at the beginning of the parallel section and sent back to the master at the end. The structure of the calculations or the middleware properly serializes and schedules the transfers so that the speculation does work perfectly (see Figure 4.7).

4.4.5 Execution Times of Application Programs

In an investigation of performance improvement that goes beyond small micro-benchmarks we looked into measuring standard benchmarks (like the NAS Parallel Benchmark [10]) on top of our new communication infrastructure as well as into full speed runs with the applications used for the traces in the previous section. Unfortunately we consistently ran into performance bottlenecks within the communication middleware for any meaningful application. NAS uses MPI, which needs a modification and a major rewrite to support

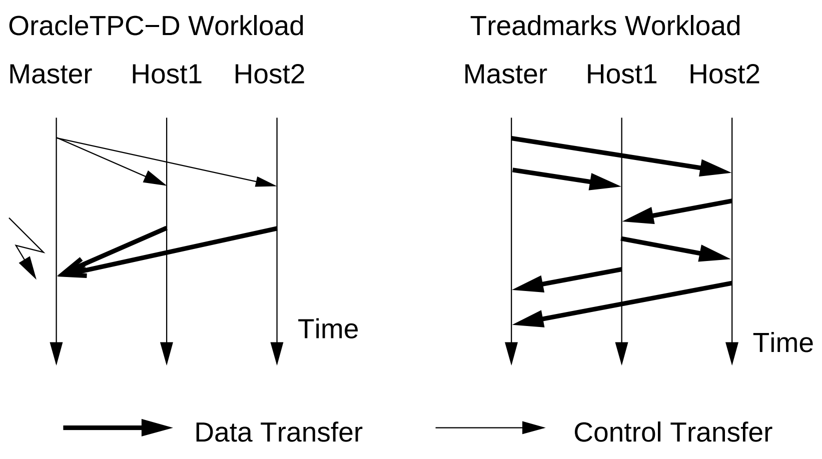


***Figure 4.7**: The Oracle workload leads to zero-copy packet interferences while in TreadMarks an inherent scheduling prevents this.*

zero-copy communications. An MPI benefiting from our new software architecture is still under construction.

The TPC-D data mining benchmark under ORACLE 8.0 results in nice traces and useful hints about traffic patterns, but we can not get that middleware to generate data at speeds higher than 1.4 MByte/s [149]. With such low communication demands Gigabit Ethernet does not matter at all and the execution times compared to Fast Ethernet remain the same.

The TreadMarks system can take some advantage of the lower CPU utilization and the higher bandwidth. Table 4.3 shows an overall improvement of the SOR example used in the last section of 4.1% (7.5% with 16 KByte pages) with Gigabit Ethernet and 5.1% (8.5% with 16 KByte pages) with zero-copy Gigabit Ethernet. The disappointing magnitude of the improvement can be explained by a communication protocol that is highly sensitiveness to latency and so a higher bandwidth does not result in better performance. With larger pages the improvement is a bit better as more data has to be communicated. Without a cleanup of the TreadMarks communication code, the gains of zero-copy communication also remain marginal because the middleware requires with many internal data copies. This looks pretty much like a chicken and egg problem as all current middlewares still rely heavily on copying semantics internally. Without a widespread Gigabit communication infrastructure in place, there is no trend in commercial software that makes use of such facilities.

As an additional application code we investigated our own data streaming tool *Dolly* [131, 132]. Dolly allows to distribute data streams (disk images) to a large number of machines in a cluster of PCs. Instead of network multicast Dolly uses a simple multi-drop chain of TCP connections. Using a logical network topology eliminates the server-bottleneck and renders a data distribution performance that is nearly independent of the number of participating nodes. Chains keep the implementation simple and reliable. The

Packet Size [KByte]	Number of Pages	Execution Times [s]		
		Fast Ethernet	Gig-Ethernet	ZC-Gig-Ethernet
4	21218	41.4	39.7 (-4.1%)	39.3 (-5.1%)
16	5509	40.0	37.1 (-7.3%)	36.6 (-8.5%)

Table 4.3: *Execution time improvements for a TreadMarks SOR code running on 3 PCs, communicating with two page sizes over different networks. Just a small performance improvement can be measured as the application is very much latency sensitive. With larger pages the improvement is higher even if more data has to be communicated.*

tool is used for distributing hard disk partitions to all nodes in widespread clusters of PCs for fast operating system installation, application software upgrades or distributing datasets for later parallel processing. In [131] we also include a performance prediction model for Dolly on different hardware and calculate the maximum achievable bandwidth of such a data distribution system by a simple analytical model. The model takes into account the limiting resources inside the nodes and predicts a maximal streaming bandwidth for a given network topology. The prediction is based on the flow of the data streams in the nodes and their usage of the resources. These limiting resources could be either the memory subsystem, the I/O bus (which is used for disk and network I/O), the attached hard disk drives or the CPU utilization.

While the absolute figure for the use of our advanced zero-copy communication architecture with real applications remains quite disappointing (see Table 4.4), the presence of a model in study [131] explains a performance improvement of 40% very accurately. The limiting factor on current PCs is the memory system utilization and so the fewer copies result in significant performance improvements and shorter down-times for maintenance in our clusters.

Dolly Streaming [MB/s]	Fast Ethernet	Gig-Ethernet	ZC-Gig-Ethernet
Modeled	11.1	11.1 (+0%)	14.7 (+32%)
Measured	8.8	9.0 (+2%)	12.2 (+39%)

Table 4.4: *Predicted and measured bandwidths for a data distribution over a logical multi-drop chain. As the CPU is the bottleneck, not the network, there is just a performance improvement when the communication copies can be eliminated. This is reflected by the model as well as by the measured performance.*

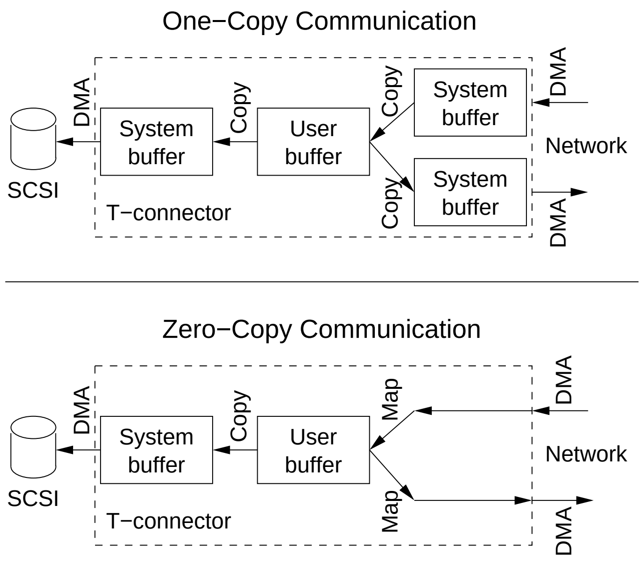


Figure 4.8*: Schematic data flow of an active node running the* Dolly *client.*

For a partition cast over a logical multi-drop chain the model predicts the same bandwidth for Fast Ethernet and Gigabit Ethernet, as it identifies the CPU and memory system of the nodes as the main bottleneck; the network is faster. If the zero-copy stack is used, i.e. the communication copies are eliminated (see Figure 4.8), the improvement due the reduced load on CPU and memories is reflected by the model as well as by the measured streaming performance. While the transition from Fast Ethernet to Gigabit Ethernet result in a marginal performance gain of 2%, there is a quite an impressive leap in performance of 39% by using Gigabit Ethernet with our new zero-copy stack based on speculative defragmentation. Since the streams of a multi-drop chain are quite static and well controlled there are close to no mis-predictions in the defragmentation process.

4.4.6 Benefit on Modern Machines

The whole development and measurements of the hardware assisted defragmentation was done on Pentium II 400 MHz PCs with a limited memory bandwidth and a limited 32 bit/33 MHz PCI I/O-bus. A look at our work in the year 2002 shows a totally different picture. For a new evaluation of the bandwidth and CPU utilization we use expensive Intel Pentium III based server systems with an improved ServerWorks Serverset III LE memory system and a much better 64 Bit/66 MHz PCI I/O-bus implementation. The memory copy bandwidth of such a machine lies at 300 MByte/s (compared to the 90 MByte/s on the old machines). We also use a newer Linux 2.4 version with an improved one copy TCP/IP stack and a much better and fine grained interrupt handling.

The much improved memory bandwidth (improvement of 3 years) already implies that it should be possible to communicate at the full bandwidth provided that the hardware can handle this. Figure 4.9 shows that the performance achievement in bandwidth of the optimized driver with speculative defragmentation is not as significant as on cheaper machines anymore (only 6%). While the original stack implementation achieves 109 MByte/s the zero-copy stack achieves 115 MByte/s which is the peak bandwidth that can be achieved given the protocol overheads.

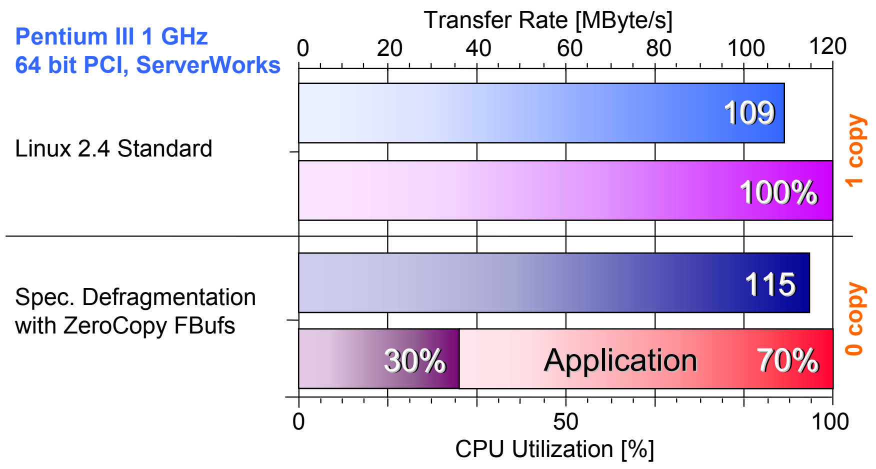


***Figure 4.9**: Throughput and CPU utilization of large data transfers over Gigabit Ethernet without optimization and in combination with speculative defragmentation and a zero-copy interface (FBufs) . The throughput is not much improved (from 109 to 115 MByte/s) as the saturation of the network is achieved. But the zero-copy stack has a much better CPU utilization of 30% compared to 100% on the original stack. The figures characterize the performance on two 1 GHz Pentium III PCs with an ServerWorks Serverset III LE chipset and a 64 bit/66 MHz PCI I/O-bus.*

However absolute speeds are only part of the performance. Comparing the CPU utilization during the transfer show a totally different picture again. Even if the network can be saturated by the original stack the CPU is heavily involved to copy and fragment the data as to well as compute the checksum. In the stack with speculative defragmentation in hardware this overhead is eliminated which results in a low CPU utilization of just 30% on a receiver. This is introduced by the receive interrupts, buffer management, TCP/IP processing.

The overlapping of communication with computation is a method of increasing efficiency and shortening the overall communication time in applications. One of the major goals in the design of parallel processing machines and algorithms is to reduce the effects of the overhead introduced when a given problem is parallelized. A key contributor to overhead is communication time. Many architectures therefore try to reduce this overhead by minimizing the actual time for communication, including latency and bandwidth. Another approach is to hide communication by overlapping it with computation. Effective use of communication hiding through overlapping communication/computation techniques can help to get the waiting time down but only if there are enough machine resources to perform communication and the computation in parallel. This is not the case with an original Linux stack as the copying utilizes the CPU to 100% and would elongate the computation phase if done in parallel.

The low receiver and even lower sender CPU utilization of the speculative stack on the other hand means that the spare cycles of the CPU can be used in the application for computation once an application is optimized for that. This allows a real overlapping of the communication and computation phases which makes these very high bandwidth transfers over Gigabit Ethernet even cheaper and more effective as they can be handled concurrently to the computation to a certain extent.

4.5 Improving the Success Rate for Speculation

The problem with a speculative solution are multiple concurrent blast transfers to the same receiver which result in interleaved streams, garbling the zero-copy frames and reducing the performance due to frequent miss-speculation about the next packet. To prevent interfering packets, we implemented a transparent admission control architecture on the Ethernet driver level that promotes only one of the incoming transfer streams to a so called fast mode. Unlike in blast facilities, this control architecture does not necessarily involve new protocols, that differ from regular IP or an explicit scheduling of such transfers through a special API.

4.5.1 Admission Control for Fast Transfers

To achieve an exclusive allocation of a host-to-host channel we have successfully implemented a distributed admission control mechanism at the driver level with specially tagged Ethernet packets that are handled directly by Ethernet driver.

A sender requests a zero-copy burst with a *Fast_Req* packet to the receiver which is answered by a *Fast_Ack* or a *Fast_NAck* packet. Although the round trip time of such a request is about 30 μs, only the invocation of the faster protocol is delayed and not the data transfer. The transfer can be started immediately with low priority and as soon as an acknowledgment arrives, the zero-copy mode can be turned on (in our software implementation this simply means that the packets are sent as three specially fragmented IP packets of 4096 Bytes in total, as described in Section 4.3.7).

In the rejection case of a *Fast_NAck* the fast transfer is either delayed or continued throttled to a low bandwidth in order to reduce interference with the ongoing zero-copy transfer of another sender. The non-acknowledge may contain the expected duration of the current fast transfer and thereby giving out a hint on when the channel may be deallocated by the fast transfer in progress. For low priority or regular transfers we use the standard

Speculation Success Rates	Oracle running TPC-D			TreadMarks running SOR		
	Master	Host 1	Host 2	Master	Host 1	Host 2
Without Network Architecture	**48%**	100%	100%	99%	>99%	>99%
With Network Architecture	**98%**	>99%	100%	99%	>99%	>99%

***Table 4.5**: Study about the rate of success for speculative transfers based on application traces without and with a driver level control architecture. The control architecture guarantees a predication rate that makes speculation worthwhile.*

flow control algorithm of TCP to slow down the bandwidth by delayed acknowledgments.

We can show in the performance analysis that such low priority data-streams do not affect the bandwidth of a fast transfer in progress even in the software solution where a re-synchronization of the descriptors has to be done for each packet which interferes a fast transfer. However it may not be a good idea to forbid extrinsic traffic at all as some of these packets may remain essential to guarantee deadlock free operation.

4.5.2 Implicit versus Explicit Allocation of Fast Transfers

Using the protocol described above, it is also possible to hide the entire network control architecture with slow/fast transfers from the user program and to implicitly optimize the end-user communication performance with standard TCP/IP sockets. The operating system or the middleware knows the size of a transfer when the corresponding system call is called. If the transfer is potentially large enough for a fast transmission, fast transfers can be automatically requested and set up so that certain data transfers benefit from a highly increased bandwidth.

As an alternative, the selection of the transfer mechanisms can be put under application control through an API. An application would only transfer the data if the fast channel were free and would otherwise try to reschedule its transfers and try to use another fast channel that is available at the time. For this option we implemented an `ioctl`-call which sends requests and returns the answers of the receivers, so that fast transfers can alternatively be put under user control. However user control is optional and the availability of an automatic implicit scheduling gives our solution a fully transparent API.

As for many large parallel applications the communication pattern of large transfers is quite regular and scheduling is done by the programmer anyway, i.e. sending MPI messages, so this mode of resource allocation in the network interface may deliver the best results.

The performance results of Table 4.2 are summarized in Table 4.5. The Oracle workload achieves nearly 100% success rate with the control architecture versus the achieved 48% without.

4.6 Enhanced Hardware Support for Speculative Zero-Copy

In Section 4.3.3 we showed that, based on speculation techniques, it is possible to implement a good zero-copy TCP/IP protocol stack with the simple hardware support available in today's Ethernet network interfaces. The speculative technique required extensive cleanup operations to guarantee correctness in the unlikely event that the defragmenter makes wrong guesses about the packet order. Speculation misses automatically raise the question of better prediction hardware to improve the accuracy of the guesses. We therefore propose three simple extensions to the NIC hardware that could greatly simplify the implementation of a zero-copy defragmenter, while preserving the compatibility with standard Ethernet IP protocols and the simplicity of off-the-shelf Ethernet hardware.

4.6.1 A Control Path Between Checksumming- and DMA-Logic

Many networking adapters do already protocol detection to help with the checksumming of the payload but this detection information can not be used to control the transfer of the frames to the host. As a first improvement to today's Ethernet NICs we propose to introduce an additional control-path between the checksumming logic with its protocol-matching engine and the DMA-Logic. This makes the protocol information available to the DMA-Logic and allows to reliably separate the headers from the payload data. In Section 4.3.3 we showed that this does not induce a totally different stack implementation, but for clients using the separation it would greatly improve the rate of success even if the support does not work for all the frames (e.g. with different TCP options).

4.6.2 Multiple Descriptor Lists for Receive

Communication at Gigabit/s speeds requires that incoming data is placed automatically into the memory of the receiving processor. At Gigabit/s speeds there is no time to take an interrupt after every incoming packet or to do a complete protocol interpretation in hardware. For zero-copy the difficult part is to deposit incoming, possibly fragmented payload directly into its final destination in memory. The previous work with ATM or VIA [157, 47] suggests that this is done on a per virtual channel or per connection basis. All known solutions require lots of dedicated hardware, a co-processor or copies to solve the problems.

Based on our strategy of speculative support we propose to divide the available descriptors into a small number of separate lists to deposit different kinds of incoming data segments directly. We suggest one list for transmitting and at least two descriptor lists for receiving, one that is usable to handle fast data transfers in a speculative manner with zero-copy and a second one that can handle the packets of all other traffic including all unexpected packets in a conventional manner. The small increase from currently two to

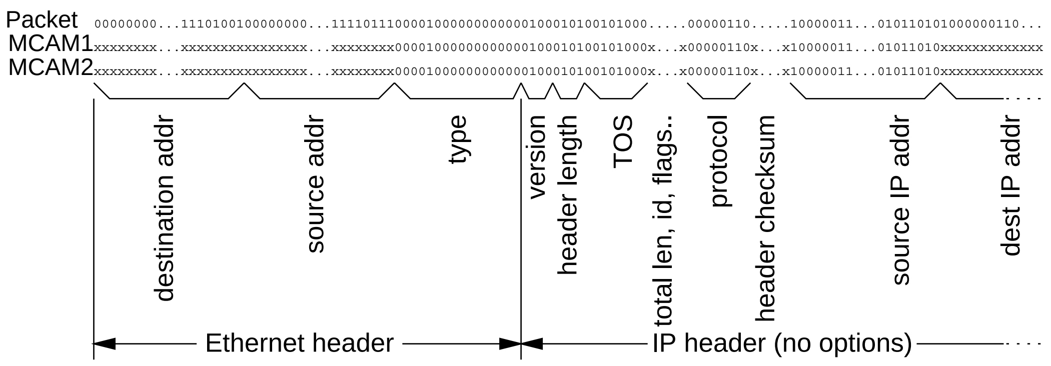

Packet 00000000...1110100100000000...11110111000010000000000000100010100101000.....00000110.....10000011...010110101000000110...
MCAM1xxxxxxxx...xxxxxxxxxxxxxxxx...xxxxxxxx000010000000000000100010100101000x...x00000110x...x10000011...01011010xxxxxxxxxxxx
MCAM2xxxxxxxx...xxxxxxxxxxxxxxxx...xxxxxxxx000010000000000000100010100101000x...x00000110x...x10000011...01011010xxxxxxxxxxxx
destination addr
source addr
type
version
header length
TOS
total len, id, flags..
protocol
header checksum
source IP addr
dest IP addr
Ethernet header
IP header (no options)

***Figure 4.10**: The bit stream of every incoming packet is matched against a content addressable match register (match cam) to detect packets to be handled by the zero-copy defragmenter. Matching against an "x" (don't care) in the match register is implemented with a second mask register. Multiple match register pairs provide associativity to match more than one protocol family. The match registers are mapped into the control space of the NIC and can easily be written by the drivers.*

three descriptor lists satisfies our requirements for a minimal change to the current adapter design.

4.6.3 Content Addressable Protocol Match Registers

The goals of using standard Ethernet switches in cluster computing is to work with mass market standard equipment for the interconnects and the switches and therefore the standard Ethernet and IP protocols must be followed as closely as possible. Messages that are handled in a zero-copy manner must be tagged appropriately. Zero-copy messages could be tagged at the Ethernet frame level, with a modified protocol ID, but this could cause problems with smart switches. We decided to use the *Type-of-Service* bits within the IP header.

As a third enhancement we propose to implement a stream detection-logic with a simple matching register to separate special zero-copy transfers from ordinary traffic.

As mentioned many networking adapters do already support a static protocol detection to help with the checksumming of the payload. But this information is not made available to control the transfer of the frames to the host and the built in detection not programmable to a specific data stream.

To keep our protocol detection hardware as flexible and as simple as possible, we suggest to include a user programmable Match CAM (Content Addressable Memory) register of about 256 bit length for every descriptor list. The first 256 bits of any incoming packet are then matched against those registers and the match is evaluated. The CAM

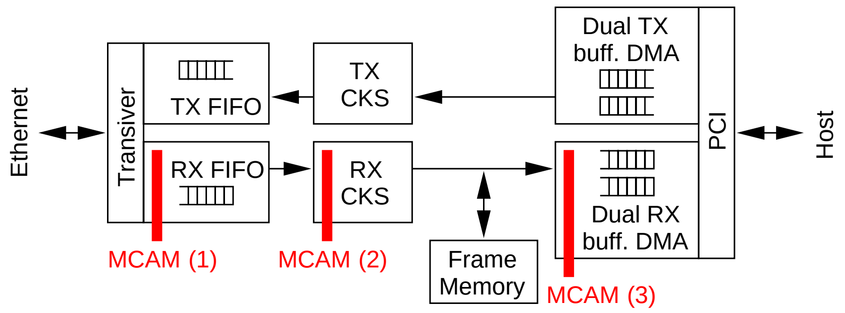


***Figure 4.11**: Schematic flow of the packets through a typical Gigabit Ethernet adapter. The gray bars indicate three possible locations (1), (2) or (3) for the proposed protocol match cam mechanism. We recommend the location (1) at the beginning of the RX FIFO since it provides the matching information as early as possible.*

properties of the match register are tailored to interpretation of protocols modified in the future and of entire protocol families. The bits should be mask-able with a “don’t care”-option and it would make sense to provide some modest associativity (e.g. 2way or 4way) for matches. An Example of a packet match is given in Figure 4.10.

The best location of this protocol matching hardware strongly depends on the VLSI implementation of the Gigabit Ethernet interface, as shown in Figure 4.11. In the most simple FIFO based designs the match operation against the first 256 bits of an incoming packet can take place at the head of the FIFO (1) while the data and the match results are propagated to the end of the queue. In the more advanced design with mandatory buffering into a staging store (like e.g. in the Hamachi chipset) the protocol match could also be evaluated when the packet is transferred into the internal staging SRAM (2) and again the result of the match would be stored with the packet. As a third option the match could be performed just before the DMA scheduler selects a packet for the transfer over the PCI bus into the host (3). The third option is probably too late since the DMA schedule needs to decide based on the availability of descriptors to transfer or not to transfer a packet to host memory. The preference of VLSI designers for early evaluation eliminates many speed path limitations, but can potentially cause coherence problems when the match registers are changed on the fly during network operations.

Match CAMs were extensively used in the iWarp parallel computers to detect and separate messages of different protocol families in its high speed interconnect. Its implementation in the iWarp VLSI component [67] consumed less than 2000 gates. Matching registers for protocol detection are already present in many NICs although they usually control only some checksumming logic and not the descriptor selection for DMA.

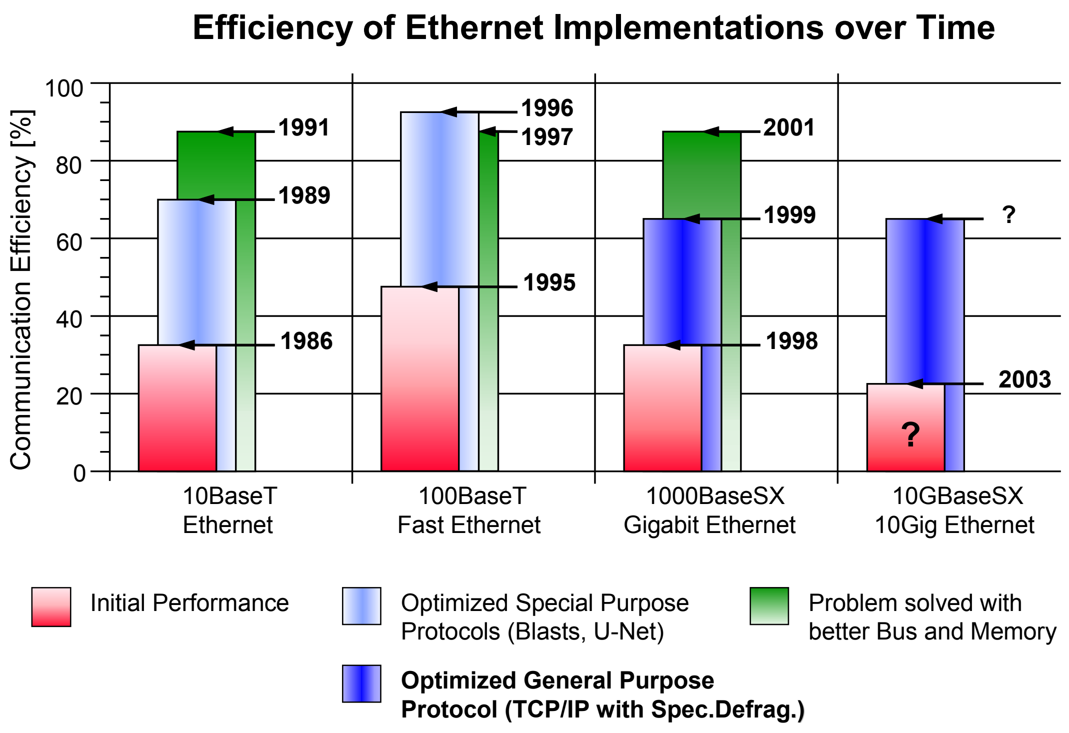


***Figure 4.12**: The hardware versus software efficiency ratios for different Ethernet implementations over time.*

4.7 Future Perspective

Before speculating about the future and the necessity of speculative driver implementation I want to look into the past and the development of the Ethernet technology and its effectiveness. Ethernet has a long history with an evolution of media from thick coaxial cables to thin coaxial cables, unshielded twisted pairs and fiber optics. It has a similar evolution from initial speeds on 1 MBit/s at Xerox Parc, to 10, 100, 1000 and 10'000 MBit/s in the current market place. Figure 4.12 shows the hardware versus software efficiency ratios for different Ethernet implementations over time. Because of the ups and downs in the efficiency we call this view the Ethernet roller coaster. Researcher work hard to pull us up to 100% efficiency like the coaster that is pulled up to the top of the first hill at the beginning of the ride. But often the same technologies are not fancy enough to reach the same 100% efficiency potential of the next generation interconnects. This is the same with the roller coaster where the conversion of potential energy to kinetic energy is what drives the roller coaster, and all of the kinetic energy you need for the ride is present once the coaster descends the first hill. But the momentum cannot be large enough to climb the same high on a second hill again.

We disregard the early prototypes at Xerox and let the roller coaster start in 1986. 10BaseT hardware interfaces were released and an initial 30% of the hardware bandwidth could be matched by the software implementations of the drivers, the communication stacks and APIs or the hardware interface of the used machines. As discussed in Section 4.2 researchers came up with the idea of special purpose protocols for blast transfers and introduced their non standard APIs to push the efficiency up to nearly 100%. But by 1991 the problem appeared to be definitely solved by better memory systems and buses in the machines and the protocols did not find their way into systems.

The next speed improvement was introduced in 1995 with 100BaseT where again not the full hardware performance could be made available to the user. At this time the Beowulf project was already well known and clustering of machines was a hot topic. Therefore special purpose interfaces like U-Net were proposed which allowed to access the hardware from user space. Even if this leads to security problems, such interfaces are used in Beowulf clusters as security is not a prime issue in this field. On top of the user level interface different low overhead protocols were implemented to achieve a very good efficiency over Fast Ethernet. In 1997 the efficiency problem was again solved by faster machines.

The same story happened with 1000BaseSX also known as Gigabit Ethernet in 1998. For the first time the network bandwidth even exceeded the memory copy bandwidth of the machines introducing an even harder pressure on researchers to come up with solutions that allow to sustain the promised bandwidths in application transfers. Of course blast transfer protocols or user level interfaces would work with this technology too. But here we came up with speculation techniques applied to communication system software. With this approach we are able to eliminate the copies and improve the efficiency of Gigabit Ethernet while preserving the general purpose interface and even the TCP/IP protocol and its readily available implementation underneath. Of course also our optimized performance can meanwhile be matched by expensive machines using a better PCI I/O-bus and a faster memory system.

The interesting question now is what the roller coaster will do further on its way. The efficiency of 10 Gigabit Ethernet (10GBaseSX) will definitely start again at an unsatisfactory level before it might be improved by better hardware support. We hope that our proposed techniques which allow the operation of general purpose APIs and protocols will find their way into optimized operating system implementations as well as adapter card hardware. We do not believe that the full 10GigE performance can be matched by better memory systems and buses so quickly. Therefore the demand for better implementations of general purpose APIs providing the promised hardware performance to the user application is bigger than ever. We think that we can accommodate this demand with our solution.

5 Zero Copy Concepts for Distributed Object Middleware

The steady decline in the price performance ratio of computing systems and the availability of high-speed networks are responsible for the rapid development and the recent popularity of distributed computing systems. As the size and complexity of application software systems is increasing, object oriented systems have attracted a great deal of attention. Both distributed processing and object oriented technology offer several benefits. Distributed systems provide concurrency in execution and performance improvement in addition to other benefits such as physical distribution, enhanced reliability and scalability. The advantages of object oriented systems, that include design flexibility and software reuse, are well known. Distributed Object Computing is a very popular paradigm because it combines the advantages of distributed processing with those of object oriented technology.

Although heterogeneity in distributed systems is natural, most distributed systems are built with only a limited heterogeneity. For example, at least some subsets of the nodes in every distributed system is likely to be similar especially in our high performance cluster computing environment. A number of inter-communicating components in an application is often implemented using the same programming languages on top of the same operating system. The components that are diverse can use an Object Request Broker (ORB) like it is defined by the Common Object Request Broker Architecture Specification (CORBA), whereas components that are built using the same programming language and operating systems can use a ”flyover”and bypass a number of standard CORBA operations and therefore save in terms of middleware overheads.

This chapter tries to address the issue of typically mediocre communication performance introduced through a CORBA ORB by proposing and implementing techniques for improving the performance of CORBA compliant middleware systems that exploit a limited heterogeneity in high performance computing systems. Zero-copy is the key technology that can be used to improve the bandwidth for inter-ORB communication.

5.1 Achieving High Performance in Middleware

The use of CORBA as a communication middleware helps the application developer to achieve flexibility and portability by automating many common development tasks such as managing object location, parameter marshaling, and object activation. CORBA is an improvement over conventional procedural RPC middleware since it supports object-oriented language features (such as encapsulation, interface inheritance, parameterized types, and exception handling) at the cost of additional overheads.

There is much literature describing research about benchmarking high-performance CORBA ORBs. As part of The ACE ORB (TAO) project [135] the latency and throughput performance of a number of ORBs, including VisiBroker from Visigenic, Orbix from IONA, and SunSoft's IIOP reference implementation was measured [65, 66]. The performance optimization of an IIOP implementation provided deeper insight into the implementation and allowed to evaluate different approaches.

The Internet Inter-ORB Protocol (IIOP) enables heterogeneous CORBA-compliant ORBs to interoperate over TCP/IP networks. The IIOP uses the Common Data Representation (CDR) transfer syntax to map CORBA Interface Definition Language (IDL) data types into a bi-canonical wire format. Due to the excessive marshaling/demarshaling overhead, data copying, and high-levels of function call overhead, conventional implementations of IIOP stacks yield poor performance over high-speed networks. To meet the demands of emerging distributed high performance computing and multimedia applications, CORBA-compliant ORBs must support both interoperable and highly efficient IIOP implementations.

5.1.1 Performance Optimizations

There are two main fields of optimizations in Object Request Brokers. In [66] the authors discuss a fully interoperable IIOP implementations and require that fine grained typed data has to be marshaled according to the rules in the specification. They pinpoint the key sources of overhead in the SunSoft IIOP implementation, which is the standard reference implementation of IIOP written in C++, by measuring its performance for transferring richly-typed data over a high speed network. They identify improvements and benefits that result from systematic protocol optimizations of the SunSoft IIOP stack. These optimizations include: optimizing for the common case; eliminating obvious waste; replacing general purpose methods with specialized, efficient ones; precomputing values, if possible; storing redundant state to speed up expensive operations; passing information between layers; and optimizing for the cache.

A second field of research for optimization targets applications running on hosts with limited heterogeneity, namely clusters of PCs, where non-typed or sparsely-typed and large amounts of similar data have to be communicated from one host to another. This is the type of performance optimization we focus on in this thesis.

In [3] the authors give three performance optimization techniques that perform bypass operations. The first technique prevents data conversion between the native data types used by system components and the standard data format specified in CORBA. The second technique saves communication time by removing the padding bytes used in between two data elements in a CORBA message that is exchanged between the nodes hosting the client and the server. The third technique shrinks the message sent by the client by compressing the message header thus reducing the communication overhead.

The first technique bypasses data conversion operations whereas the last two bypass the transmission of certain bytes in the messages exchanged between a client and its server. The degree of performance improvement depends, however, on the nature of the application workload. For the multimedia and high performance applications huge amounts of data have to be moved from one host to another and the last two optimizations have a very marginal effect on the performance, because only a few bytes in the headers can be spared. We therefore concentrate our efforts to bypassing of the conversion which again is one of the main overhead in todays ORBs. Getting rid of the expensive copying operations in the marshaling/demarshaling can improve system performance significantly.

We will therefore try to put the system under a zero-copy regime. Again the aim is to hide the optimization techniques and incorporated them into the middleware so that the interface stays transparent to the application program.

5.1.2 Optimization Incorporation

The technique of bypassing a marshaling routine requires the modification of the message buffer contents as well as a modification of the data path through the ORB for achieving a more efficient data communication. CORBA defines a number of standard interfaces for the ORB. These interfaces are used at the client side to dispatch requests to servers. The contents of the client buffer are received in an input buffer on the server side ORB interface. The performance improvements require changes at both buffering stages, the output buffer at the sender and the input buffer at the receiver.

There are two approaches for incorporating the optimization techniques in an open source ORB: (a) Interceptor programming and (b) Modification of source code. In the subsequent paragraph we describe advantages and disadvantages of both options and argue for the modification of source code in our case.

Interceptors are approach to change the standard behavior of an ORB by providing a flexible means for having custom code called by the ORB at specific points during operation invocations. By using interceptors one can interact with a CORBA transaction, monitor and log method calls, or define simple access control policies. There are several points at which interceptors can be invoked during the processing of a CORBA transaction. Interceptors are objects of particular classes, which are called at specific points during the processing of a CORBA transaction. By defining subclasses of interceptor classes and registering instances with the ORB, one can insert ones own code into the invocation path. This allows access to the processing of the request itself and provides a powerful way of observing and/or modifying requests. On the other hand, it requires more detailed knowledge about ORB data structures than necessary for writing application level code.

A CORBA ORB is typically implemented as a run-time library linked into both client and server applications. If there is direct access to the source code of the ORB library, the routines that perform marshaling/demarshaling on the data structures can be changed or rewritten. There are a small number of Open Source CORBA ORBs available today such as MICO, omniORB, Orbacus or jacORB. These packages provide the user with the source code of the whole ORB implementation, which can be altered and recompiled, thus allowing a change in the formation of the output data at the sender side and its interpretation at the receiver side. We use this approach in performance enhancement strategy for this chapter.

Experimental investigations on our cluster of Pentium II PC's connected by a Fast Ethernet and a Gigabit Ethernet running under Linux indicate that performance optimization techniques can lead to a significant improvement in system performance.

5.2 Limited Heterogeneity

CORBA has proven its worth in low speed networks that do not demand high system performance. It provides a flexible solution for a heterogeneous environment, which is a characteristic of 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, embedded 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 [135]. As a result e.g. many 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 [2].

Heterogeneity is natural in many distributed systems. Adding new components to an existing system in terms of added functionality or system resources often leads to diversity in hardware, operating systems, and programming languages used in implementing these components. Middleware provides the inter-operability that is necessary for these diverse components to inter-communicate with one another. Using such a middleware product, it is possible for a component of the application to communicate with another component that is implemented in a different programming language and is running on top of a different platform. The Common Object Request Broker Architecture is a standard for distributed object middleware with a rapid growth of interest. But many concepts are similar to other solutions and optimized implementation techniques can be equally be applied to these standards as well.

Although heterogeneity in distributed systems is natural, most distributed systems are characterized by limited heterogeneity. For example a subset of the nodes in the system are likely to be similar especially in our cluster environment, where we even count on totally equal systems. Also most of the inter-communicating objects are implemented using the same programming languages on top of the same operating system. Improving the performance of CORBA compliant middleware systems that exploit limited heterogeneity in systems is especially in clustered environments vitally important.

5.2.1 Standard Data Path Bypassing

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 [134]. Since we focus on large data transfers where demultiplexing and memory allocation is not a limiting factor the data copying overhead is most interesting for us.

There are different approaches how expensive data handling can be bypassed. In Section 5.1.2 we already explained two ways to incorporate that well known bypass into an ORB. The following sections explains three totally different ways on how to implement such a bypass.

The knowledge of all those previously known techniques is required but still not sufficient for the implementation of a zero-copy data path in an ORB.

5.2.2 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 [86]. PARDIS is an environment providing support for building parallel distributed applications. It employs the key idea of CORBA — interoperability through meta-language interfaces — to implement application-level interaction of heterogeneous parallel components in a distributed environment. Addressing interoperability at this level allows the programmer to build meta-applications from independently developed and tested components. This approach allows for a high level of component reusability and does not require the components to be reimplemented. Further, it allows PARDIS to take advantage of application-level information, such as distribution of data structures in a data-parallel program.

PARDIS builds on CORBA in that it allows the programmer to construct meta applications without concern for component location, heterogeneity of component resources, or data translation and marshaling in communication between them. 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 PARDIS Object Request Broker (ORB) — the entity responsible for brokering requests between clients and servers. This capability ensures request delivery to all the computing threads of a parallel application and allows the ORB to transfer distributed arguments directly (if possible in parallel) between the client and the server. PARDIS uses a simple activation model for threads. Single objects are always associated with just one computing thread and they are also supported and can be collocated with SPMD objects. In addition, PARDIS contains programming support for concurrency by allowing non-blocking invocation returning distributed or non-distributed futures, and allowing asynchronous processing on the server's side.

5.2.3 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 [96] describes 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. Performance comparisons for a few problem solving environments for molecular dynamic simulations that compare between runs of the CORBA object and the original legacy code on a cluster of workstations and on a parallel computer showed similar performance results.

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. The advantage is that users can use existing CORBA implementations without any modification to CORBA IDL compilers, as is done in other projects with a similar objective, such as PARDIS [86] and Cobra [127]. Nevertheless with wrapped legacy codes it remains unclear if the advantages of object orientation are included in the system.

5.2.4 Bypass of Marshaling/Demarshaling

The original idea behind this 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. The standard path for the code of the object communication would be: the client calls a stub and passes the parameters on the stack; the code of the stub marshals the parameters from the stack into a shared memory area; the server skeleton demarshals the parameters from shared memory onto the stack; and passes the control to the implementation; and then the other way back.

By bypassing the marshaling also the data copying within the middleware can be bypassed. With compiler support, one could even have the application push the parameters directly into the stack in a shared memory area and have the implementation work directly from there, but we don't want to use special compilers. The alternative is using a set of macros and/or inline functions, possibly in combination with a function stub.

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 chapter.

5.3 The Underlying Communication Unfrastructure

A messaging subsystem is part of all parallel or distributed systems and transfers data from one process or thread to another. 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.

Even if CORBA mainly implements a synchronized client server paradigm that allows remote object invocation, the techniques used inside the middleware are mainly the exchange of messages. We therefore look deeper into the services of what a much more general purpose message passing environment provides to gain hints with what techniques buffering can be eliminated in such systems. We then study the resulting techniques and try to integrate them into the ORB design to optimize CORBA middleware.

For message passing communication 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 location of the data is in the local memory of sender 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 local memory at the receiver. In the simple message passing model for each message transfer a send call must conceptionally match a single corresponding receive call.

The originator of the transfer is not restricted to the sender node. With appropriate hardware or software support a destination node may initiate the transfer and fetch some data out of the sender's memory, or vice versa the sender might deposit some data into the receiver's memory and even complete the whole data transfer without the receiver's participation. The difference between a classic send/receive and a fetch/deposit transfer is in the synchronization and the control transfer provided along with the data transfer.

Most descriptions of message passing systems are purely based on the mechanics of data transfers and miss the important point of the synchronization that goes along with every data transfer. A data transfer can take place under a variety of synchronization assumptions and the programmer must be well aware of the control transfer semantics to write correct parallel programs that are free of data races and free of deadlock.

5.3.1 Message Passing Models

Every message passing library comes with a set of assumptions and rules to write correct programs. Such a set of a assumptions is called a message passing model. The two most common models of message passing are the rendez-vous model and the postal model.

In the *rendez-vous model* every data transfer must use a two-way handshake to ensure a proper meeting between the two communicating processors for any data exchanged. The model requires total control over the communication schedule in the distributed computation. The simplicity and the rigor of the model made it popular for theoretical work. In its strict form the classic rendezvous model deals with atomic sends and receives.

The *postal model* of message passing incorporates buffering into the basic message passing services and provides a one way synchronization with each data transfer. Its definition and use is simple and highly appealing to programmers. The receiver can receive the data immediately or at a later time. If the receiver does not claim the data immediately, it must be stored in the network or a temporary location until the receiver is ready to accept the data. The presence of a temporary buffer (mailbox) and the features of the well known postal system to delay and store an almost unlimited amount mail in transit results in the intuitive name. The consequences of a one way synchronization are obvious: No receive operation can complete before the corresponding data (i.e. the matching message) has been sent. But conversely the send can terminate regardless whether a receive is ever invoked or not. Buffering must take care of the situation of a send with a receive at a later time.

5.3.2 Control and Data Transfer Messages

The basic message passing models can be defined with a single type of messages. For the extension of this basic models to the deposit model a precise distinction between control and data messages is required [144].

All messages can be classified based on their content, their length and their purpose and are separated into two classes: control messages 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. The meaning of such an assertion is, that some data block is ready to be transferred, that some buffer is available to receive more data or that some previous data message was or was not transferred successfully to its destination. In the latter case the reception of a positive or negative acknowledgment automatically initiates retransmission or flags an error in a communication system that supports reliable transfers.

All implicit messages generated by the lower protocol layers of a message passing stack are classified as control messages. Even 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.

The sole purpose of control messages is to communicate program state and implied assertions between processors in a parallel or a distributed system.

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.

5.3.3 Decoupling Synchronization and Data Transfers

The separation of synchronization and data transfer is the key to communication performance in parallel computers [145]. Fully decoupled and independent synchronization generates many opportunities for improvement and optimization of both the synchronization and the data transfer component of a communication system. Optimizations 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 if a middleware establishes the communication to this software in between. Looking at the structure of CORBA applications this means that the buffers are allocated and managed by the application or the stub and skeleton code generated through the toolkit of the according ORB.

For scientific applications written in parallelizable languages high performance parallelizing compilers are aware of the global information about the communication pattern. That eases the code generation for buffer management in compiler generated parallel programs. Based on the same knowledge the compiler can insert additional synchronization primitives until the generated code qualifies as a well-synchronized program. Once the program is well-synchronized its data transfers no longer require buffering and the buffering mechanism are eliminated.

As with CORBA in many scenarios the buffer management is handled by the middleware it becomes possible to optimize this buffer management in the middleware implementation. This does not effect the user application while allowing much faster communication. Instead of using parallelizing compilers which have knowledge about the communication pattern of a user application we rely on the knowledge that we as a programmer have about the communication pattern of the ORB. To avoid changes the application interface and the synchronized client server messaging model of CORBA we introduce a decoupling of synchronization and data transfer within the IIOP communication system of the ORB.

5.3.4 Decoupling Increases Performance

According to [145] there are four ways in which the separation between control and data transfers can simplify message passing systems and improve performance. This is again seen in the environment of parallelizing compilers and supercomputer hardware and we have to apply the same technique to distributed systems and clusters.

Decoupled message passing uses this separation to:

Target data directly to its final destination Synchronization is required prior to data transfers if we want to target the data directly to its final destination (which is always desirable to avoid copying).

Put buffers under user control In communication systems without system buffers, all data transfers are fully under user control (desirable to avoid copying). Therefore we cannot rely on any synchronization provided by data transfers. A separate synchronization mechanism and/or a global synchronization concept is required.

Eliminate the need for buffering If synchronization and data transfers are coupled, a combined control and data message may involve buffering and costly storage management operations, since these are necessary for data transfers to complete. With prior synchronization we can be sure that there is no buffering and eliminate all the housekeeping overhead for buffering in the message passing system.

Combine control messages into global communication primitives If synchronization and data transfers are decoupled, control-only messages can be combined into cheaper global operations like hardware barriers. This algorithmic optimization reduces the message complexity of many flow control protocols from $O(P)$ to $O(log(P))$. P is the number of processors.

Message passing models with decoupled synchronization and data transfers can simplify the design of message passing libraries and permit the exploration of novel mechanisms for data transfers and new communication optimizations.

Applying these optimizations to a CORBA ORB will allow us to optimize the buffers within the ORB to that effect that the buffering can be eliminated. That means the data is stored directly into the destination buffers. As optimal support for synchronization is mainly a question of latency and special purpose hardware and is not an issue in CORBA applications we cannot address the hardware issue working with commodity systems.

5.3.5 Direct Deposit Messaging

In the Direct Deposit messaging system [145] decoupled synchronization is used and all messages are taken directly from memory (user space) at the sender and are automatically directed to their final destination in the memory at the receiving end. While direct deposit is not coherent shared memory it resolves the issue of addressing the data at a fine granularity and permits transfers of a collection of data elements directly from their source locations to their final locations. If temporary data structures or buffers are used, they are under middleware, compiler or user control and synchronization messages are generated separately. Like all receive-less communication mechanisms, direct deposit dumps its data blindly and cannot establish any synchronization properties with its data transfers. There exists a possibility of live data being overwritten and the receive mechanisms can not address the problem. A decoupled form of synchronization is needed for correct execution and for data consistency across the processing nodes.

The synchronization model does not determine how the address information is specified for the transfer; it merely specifies that the final address for the transferred data is known at the time of the actual data transfer across the network. Multiple solutions to the problem of the address translation remain possible.

A further dimension in the space of addressing modes is whether the addresses associated with the data are physical, global virtual, or local virtual. Most advance network adapters that allow direct deposit via DMA require that all memory pages involved in this process are pinned into physical memory beforehand.

We will use the term direct deposit in this chapter synonymically with zero-copy transfers with decoupled synchronization. This implementation technique is exactly what we intend to introduce into a CORBA ORB.

5.4 Connecting Client and Server

Much of the initial success of CORBA was due to the early standardization of its object invocation protocol, the General Inter-ORB Protocol (GIOP). The Internet Inter-ORB

Protocol (IIOP) is generally a mapping for GIOP over TCP/IP. This lead to the most widespread deployment in the TCP/IP environment.

Although GIOP was originally envisaged as a protocol for communication between CORBA ORBs from different vendors, each vendor would internally use its own proprietary protocol between two objects managed by the same ORB. Today, GIOP is a widely deployed protocol and is increasingly being used in areas beyond CORBA. As an example the Java Remote Method Invocation (Java RMI) technology runs over Internet Inter-ORB Protocol (”RMI-IIOP”) [146]. This capability was introduced to the Java 2 platform and was developed by Sun and IBM.

Although GIOP is a well-designed and widely implemented protocol, it is not always implemented efficiently. This is particularly true in the case of commercially successful GIOP-based ORBs which, according to the literature [68], are all significantly slower than research ORBs such as TAO [135], omniORB [97], or GOPI [30]. The main reason for this performance deficit is that the commercial vendors, quite understandably, have traditionally focused on full capability and reliability rather than on performance.

There is an increasing demand for GIOP-based ORB technology in more demanding areas such as interactive, multimedia and mobile systems and to deploy ever larger and more complex distributed applications. In such environments high performance becomes crucial. The authors of [31] discuss an efficiently implementation of the GIOP protocol in the context of the GOPI high performance research ORB.

5.4.1 The GIOP Protocol

The OMG’s GIOP specification [112] comprise three distinct standards [133]. First, a messaging standard defines packet headers, protocols for remote communication, and requirements on the underlying transport service. Second, the Common Data Representation (CDR) standard defines on-the-wire encodings for primitive and structured data-types in messages. Finally third, the specification defines the structure and content of Interoperable Object References (IORs) which act as location transparent object identifiers. In this section, we focus exclusively on the messaging standard and its optimal implementation. The IOR part of the specification are not discussed further because issues relating to their implementation are not addressed in this dissertation.

5.4.2 Common Data Representation

In order to provide interoperability in a heterogeneous environment, CORBA has defined a standard or neutral way to transfer data, a so called bi-canonical, on-the-wire data representation named Common Data Representation (CDR). This is because different computing nodes may represent various data types using different internal or native data representation. This allows the decoupling of knowledge about architecture. GIOP defines a CDR that determines the binary layout of IDL data types for transmission. This data is aligned on its natural boundaries within a GIOP message. The CDR represents all data types available in IDL. This means, that IDL developers do not have to worry about marshaling any of their own data types.

5.4.3 Marshaling

Differences exist in the byte ordering of words on different machine architectures. These architectural inconsistencies result in multiple machine data addressing domains. Specifically, different architectures have different views of the most and least significant bytes. Therefore the data must undergo some sort of transformation process before transmission. Sending the data as is would cause erroneous results if the sender and receiver had different byte ordering.

Most machines are byte-addressed and provide access for bytes, halfwords, words, and double words. Typically, there are two ways that a machine orders bytes within words: big endian and little endian. GIOP allows to tag a message with a flag to indicate endianness and then allowing the sender to send the message in its native endianness.

GIOP further specifies primitive data types that are encoded in multiples of octets. This means that data types always use the same number of octets to represent their values. Complex types, such as structures, are built from the primitive types.

5.4.4 CDR and Streams

When a client invokes a distributed operation on a server, it transfers the operation data it is sending to the server as an octet stream. The CORBA specification defines an octet stream as “an abstract notion that typically corresponds to a memory buffer that is to be sent to another process or machine over some IPC mechanism or network transport.” An octet is an 8-bit value that undergoes no marshaling, either by the client or by the server. An octet stream is a sequence of these octets that is arbitrarily long and that has a well-defined beginning. An octet does not undergo conversion from one byte order to another; the transmitter and receiver leave it as is. All data must undergo marshaling before insertion into the octet stream.

Operation data that has been streamed no longer has a relationship to its original data type; it is simply a sequence of octets. For its content to be understood, a standard set of transforming rules must be applied to this sequences of octets. CORBA defines these rules in the CDR transfer syntax for the formatting of the OMG IDL data types in the octet stream.

There are two kinds of streams: messages and encapsulations. The *message* is GIOP’s basic unit for information exchange, while the *encapsulation* is an octet stream into which the marshaling of IDL data structures may occur separate from any message.

The representation of encapsulated data structures occurs as an octet sequence; in IDL, the data type `sequence<octet>` allows a marshaled octet sequence to be added into another encapsulation or message. The encapsulation allows the premarshaling of complex constants such as type codes, thereby saving the overhead of having them marshaled at runtime. It also can contain other encapsulations and can handle these without demarshaling them. This is particularly interesting when focusing on zero-copy optimizations. As discussed earlier with zero-copy optimizations we concentrate on large data blocks that have to be transfered. These superbly match with sequences of octets.

The encoding of a sequence starts with its length, encoded as an `unsigned long`. The elements of the sequence follow, encoded as their type. In the following an unbounded sequence of octets is defined and created. This is the type we will study and optimize in our implementation:

```
typedef sequence<octet> octetSeq;
octetSeq mySeq;
```

Setting the length to 5, and filling it with the values 1 through 5 produces the following stream:

```
0x00 0x00 0x00 0x05 0x01 0x02 0x03 0x04 0x05
```

5.4.5 Transport Requirements

The design of GIOP does not rely on a specific transport protocol. Instead, it allows the use of a wide range of transports facilities. However we concentrate on TCP/IP because of its popularity and its specification as the baseline transport protocol for GIOP. GIOP fits naturally into TCP/IP because it makes several assumptions about its transport layer. These are:

Connection-oriented Transport GIOP uses connection IDs to map requests to replies in communications between client and servers. The definition and lifetime of these IDs is within the scope of the connection. Once the connection terminates, the IDs are no longer valid. GIOP therefore depends on connection-oriented transport such as TCP. So a sent datagram does not require acknowledgment.

Reliable Transport GIOP does not have the facilities to check for packet ordering, bad packets, lost packets, or duplicate packets. It therefore relies on the transport protocol.

Stream of Byte Transport There can be no arbitrary message size limitations enforced. The transport protocol may not require GIOP to fragment or byte-align the data. The fragmentation of large messages is handled by the transport layer.

Connection Loss Notification When a disorderly connection loss occurs, e.g. when a client crashes, the transport layer must give the server some reasonable notification of this event.

Connection Initiation Model The transport layer's model for initiating connections has a mapping onto the general connection model of TCP/IP. The server is not the connection initiator. Instead it waits, prepared to accept a request from a client to connect. In TCP/IP terms, it listens for connections. A client must know the address of the server before it can initiate a connection.

5.4.6 GIOP Messages

Once a connection is established the client and server may begin sending GIOP messages between each other. In the most recent form, the GIOP messaging standard defines an object request protocol that incorporates eight message types: Request, Reply, LocateRequest, LocateReply, CancelRequest, CloseConnection, MessageError and Fragment. Figure 5.1 lists these messages and their relationship with the client and server. Each message must have a GIOP header that identifies the message and its byte ordering.

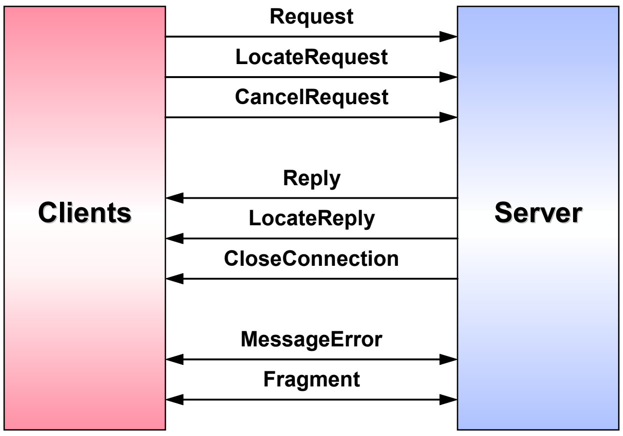


***Figure 5.1**: GIOP Messaging*

The eight messages are sufficient to accomplish the most complex of distributed tasks. They fall into two categories:

Administrative messages These are the LocateRequest and LocateReply messages used to find objects, Cancel Request and CloseConnection used to handle requests that are taking too long to execute or that are no longer desired and MessageError, sued for error handling.

Object Invocation These messages are Request, Reply, and Fragment, which are used to request an operation on an object and to allow the object to reply.

GIOP is a client-server protocol. Request messages, which carry all the information necessary to invoke a remote object, are sent by clients, and Reply messages, which are sent in response to Request messages, are sent by servers. Client and server roles (respectively) are similarly assigned to the LocateRequest and LocateReply messages; this pair is used to query the current location of an object. It is permissible to multiplex requests on connections; i.e. one can issue new Request (or LocateRequest) messages on a given connection before replies to previously issued requests on the same connection have been received.

The remaining four messages are self-standing rather than paired. CancelRequest, a client-side message, is used to advise servers that a reply is no longer required for the (still pending) request whose identifier is specified in the message. CloseConnection is a server-side message used to advise the client not to send further requests on the connection on which the CloseConnection message was received, as this connection is about to be closed. Finally, MessageError and Fragment messages can be sent by either clients or servers. The former is sent in response to any message with a bad header, and the latter is used to support multi-fragment messages. Fragment messages follow an incomplete preceding message (of type Request, Reply, LocateRequest, LocateReply or Fragment) which has its following fragment bit set. The last Fragment message in a multi-fragment message has its following fragment bit unset.

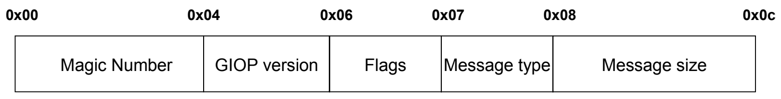


Figure 5.2: *GIOP Message header*

All GIOP message types employ a fixed-sized message header (Figure 5.2). The magic field in this header is used to identify messages as GIOP messages, the version field specifies the GIOP protocol version and the message type field identifies the message's type (i.e. as one of the eight possible types described above). The flags field includes a bit to specify whether the sender is running on a little or a big-endian architecture and also a following fragment bit to specify whether or not this message is complete or only a fragment. The message size field contains the length of the whole message in octets, excluding the 12 octets of the fixed-sized message header itself.

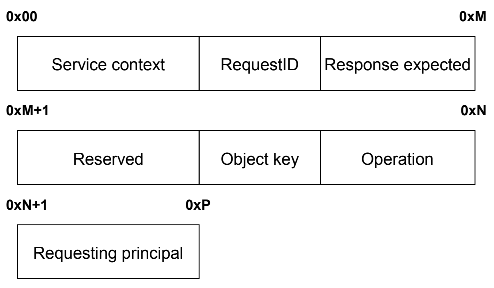


Figure 5.3: *GIOP Request header*

In addition to the fixed-sized message header, all message types except CloseConnection, MessageError and Fragment additionally employ a message specific header situated between the message header and the payload (Figure 5.3).

For example, the Request message's specific header comprises the following sequence of fields:

- an unsigned long l followed by a list of l *service contexts* (l is 0 for an empty service context list); service contexts contain auxiliary information (e.g. a transaction or security identifier or priority information) that may need to be passed to an operation invocation; they are each encoded as an unsigned long m followed by m octets of data;
- an unsigned long containing a unique *request identifier*; this is used to match Requests with their corresponding replies and to identify Requests in CancelRequest messages;
- an octet interpreted as a boolean that specifies whether or not a response to this Request is expected;
- three octets that are currently unused but serve to pad the previous field so that the following field is appropriately aligned;
- an unsigned long l followed by l octets representing the *object key*; this is the unique identifier of the target object;
- an unsigned long l followed by $l - 1$ octets representing the *operation string*, followed by a null octet; the operation string identifies the target operation name (this is assumed to refer to an operation supported by the target object);
- an unsigned long l followed by l octets representing the *requesting principal* (l is 0 for empty requesting principals); this field identifies the requesting object (e.g. for security or accounting purposes).

The fields comprising the message-specific headers of the other message types are largely subsets of the above set of fields. More specifically, the fields of the remaining message types are as follows. Reply: service context, request identifier and reply status (the latter is an unsigned long); LocateRequest: request identifier and object key; LocateReply: request identifier and reply status; CancelRequest: request identifier (of the request to be canceled). In the LocateReply case, if the reply status field indicates success, the message payload is assumed to contain a marshaled IOR that specifies the current location of the queried object.

Note finally that the CancelRequest and LocateReply messages employ fixed length message specific headers whereas the Request, Reply and LocateRequest headers are of

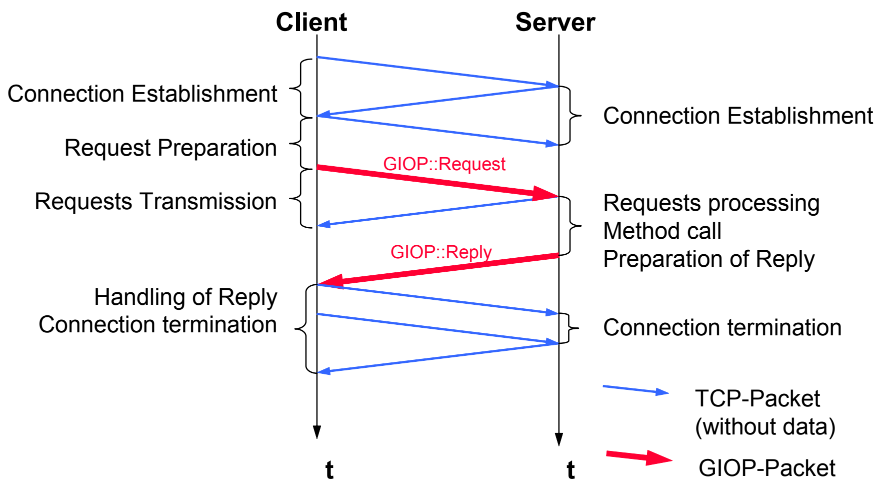


Figure 5.4: *Remote method invocation with IIOP*

variable length because they include variable length fields (i.e. one or more of: service context, object key, operation string or requesting principal fields).

5.4.7 The IIOP Protocol

The Internet Inter-ORB Protocol (IIOP) is the most widely known of the standards from the Object Management Group (OMG).

GIOP defines a protocol that is independent of any particular set of network protocols, such as IPX or TCP/IP. Given the explosive growth of the Internet, the most common networking protocols have become TCP and IP. IIOP is the protocol adopted by the OMG that must be supported by CORBA-compliant networked ORBs, either as a native protocol or through half bridges. Essentially, IIOP is just a mapping of GIOP onto the Internet's TCP transport layer. Mappings onto other transport layers may be defined in the future.

An ORB may support optional ESIOPs (Environment-Specific IOPs) as its preferred ORB protocol. DCE-Common IOP (DCI-CIOP) is the first such protocol that has been publicly specified. It uses a subset of DCE-RPC facilities and reuses parts of GIOP.

IIOP compliance requires that agents that are capable of accepting object requests or providing locations for objects publish there TCP/IP addresses in IORs. Any client needing the published object's services initiate a connection with the object, using the address specified in its IOR. The agent may accept or reject connection requests.

As explained GIOP is insufficient for enabling client-server interaction. IIOP is the glue that binds GIOP and TCP/IP. An examination of TCP/IP shows that it falls short in several areas. The first is in providing a consistent language that allows clients and servers to speak. The second is in defining a standard method to allow clients to discover services. By discovery is meant not dynamic discovery in the sense of finding objects through the naming service, but rather what information must be in a handle or reference in order for clients and servers to communicate. For a client to communicate with an object, it must have a reference to it. This object reference in IIOP is the IOR, which contains a host address and a port number. A client wanting to invoke on the object can send a Request message to the host and port listed in the IOR. The object that is the target of the invocation lives within a server process in the host machine. The server listens at the port for requests and, when they come in, dispatches them to the object. This scenario requires that the server is always running and actively listening on the port for requests.

IIOP was mainly created to address the particulars of the TCP/IP protocols. Specifically it provides host addresses through IP addresses and machine process addresses through port numbers. IIOP was kept very thin, providing only what GIOP needs in order to establish a connection. Figure 5.4 shows the communication introduced between a client and a server by calling a method of a CORBA object. First the connection is established by according to the data in the IOR provided to the IIOP, then the request and reply is handled by the GIOP and afterwards the connection is terminated again.

5.5 MICO as a Software Platform for Zero-Copy

This section first measures the performance of the MICO ORB and then gives a detailed analysis where the overhead inside the middleware occured.

5.5.1 MICO our Software Platform

To study the performance of CORBA and implement our optimizations we used one of the large and sophisticated open source projects around. The acronym MICO [130] expands to MICO Is CORBA. The intention of this project is to provide a freely available and fully compliant implementation of the CORBA standard.

MICO has become quite popular as an open source project and is widely used for different purposes. 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. The goal is to keep MICO compliant to the latest CORBA standard. The sources of MICO are placed under the GNU-copyright notice.

MICO is implemented in C++ and includes many features as: IDL to C++ mapping, Dynamic Invocation Interface (DII), Dynamic Skeleton Interface (DSI) , IIOP as native protocol (ORB prepared for multi-protocol support) , Portable Object Adapter (POA) , Objects by Value (OBV) , CORBA Components (CCM) , Interceptors , Support for secure communication and authentication using SSL and many CORBA Services like Interoperable Naming service, Trading service , Event service.

5.5.2 Shortcomings and State of the Art in CORBA Middleware

At this point we first show the bandwidth performance of an unoptimized socket API as well as the bandwidth achieved by MICO. We therefore use the TTCP benchmark described in Section 5.7.1 as well a corresponding CORBA version introduced in Section 5.7.2. The hardware platform (see Section 5.7.3) consists of the same cluster nodes used in Chapter 4. Figure 5.5 summarizes the TTCP benchmark performance results for the benchmarks over Gigabit Ethernet. The single measurement results show very little variation. It is clear that the CORBA-based TTCP implementation ran considerably slower than the raw TCP version programmed in C especially on Gigabit Ethernet but even on Fast Ethernet. The CORBA performance for all tests is poor and lays around 50 MBit/s even if the raw TCP socket version without zero-copy can achieve 330 MBit/s. The potential for a zero-copy optimization is therefore huge.

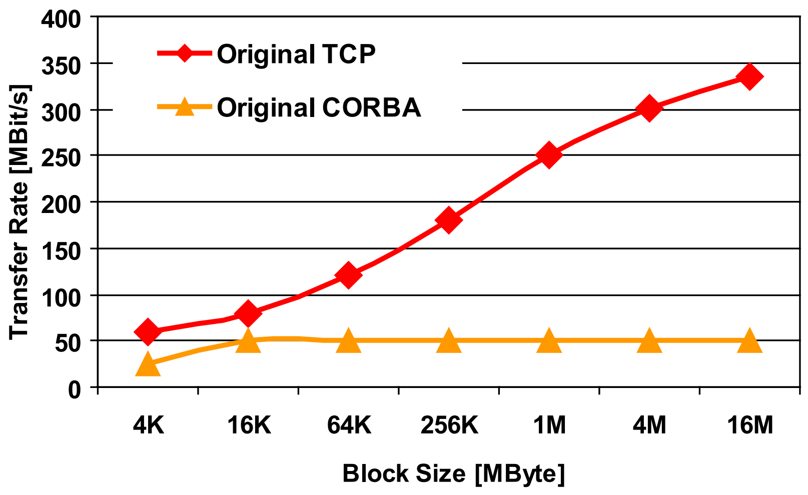


***Figure 5.5**: Bandwidths measured with TTCP for sockets and CORBA. Both tests were run without zero-copy optimizations.*

Since the data that is sent is untyped the CORBA presentation layer would not need to perform complex marshaling to handle byte-ordering differences between sender and receiver. But although marshaling is not required, the CORBA implementations incurred significant data copying overhead.

We instrumented the ORB source code and used the UNIX execution profiler `prof` to pinpoint the sources of this overhead. The C++ compiler was directed to instrument the source code with monitoring instructions and `prof` was then used to measure the amount of time spent in functions during program execution.

The functions where the most time was spent when sending and receiving 16 MByte while using 4 KByte data buffers and 64 KByte socket queues where the read and write system calls. They accounted for more than 98% of the execution time in the raw TCP C-implementations of TTCP.

Although the data was transmitted as 4096 separate 4 KByte buffers the receiver often read much smaller chunks of around 1-2 KByte. This illustrates the fragmentation and reassembly performed by the TCP/IP stack for the Ethernet (whose MTU is 1500 Bytes).

The read and write system calls had quite an influence on the execution of the CORBA implementations as well. Unlike the C versions, however, these implementations spent 70% percent of their time performing other tasks, such as copying and/or inspecting data (`memcpy`, `strcpy`, and `strlen`), checking for activity on other I/O handles (`poll`), and manipulating signal handlers (`sigaction`).

The highest cost tasks involved data copying and data inspection. Even if the copy could have been a contiguous block copy, MICO uses a very general unoptimized copy loop that is able to handle many different types. Even if the copy is performed only once it dominates the overhead. And note that there is another copy in the standard TCP/IP stack which is the main reason for the `read()` and `write()` overhead.

Another source of overhead is the memory allocation of the IDL skeletons. These are generated automatically by a CORBA IDL compiler and normally do not know how the user-supplied upcall will use the parameters passed to it from the request message. Thus, they use conservative memory management techniques that dynamically allocate and release copies of messages before and after an upcall, respectively. These memory management policies are important in some circumstances (e.g., if an upcall is used in a multi-threaded application). However, this strategy needlessly increases processing overhead for streaming applications like TTCP that consume their data immediately without modifying it.

The MICO implementation of CORBA is based on C++ where a modified version of the STL (Standard Template Library) builds the foundation for the handling of data structures. The analysis of MICO includes the ORB itself as well as the IDL-generated stub and skeleton for the test application. In the analysis we focus on the intern handling of the data parameters passed by a CORBA method invocation.

As mentioned before we used a step by step instrumentation of the MICO source code to generate debug output.

There are two facts that made the analysis awkward and delicate. There is first the deep inheritance hierarchies used in the ORB implementation. This often leads to many possible candidates of implementations for a method call under study. Secondly the authors excessively use the operator overloading technique. Functionality made available by overloaded methods made it very difficult to find the real implementation in the code as the method calls hide behind the normal language syntax.

In [66] a similar analysis is done for Orbix and ORBeline. The authors found the same performance bottlenecks but had even higher performance variations between Orbix and ORBeline. They found that this results from differences in the message fragmentation/reassembly implementations of the tested ORBs, as well as the design of their socket event handling. ORBeline copies data approximately 3 more times than Orbix on the sender and receiver for sequences.

5.5.3 Data Structure Analysis

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 key to them which consists of an integer value called *Type Identifier* (TID).

Of main interest for this chapter is 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 sequence of these octets is called an octet stream. It is arbitrarily long and has a well-defined beginning. A sequence of octets does not undergo conversion from one byte order to another either; the transmitter and receiver leave it as is. A second interesting point in the semantic of streams is that CORBA defines an access method that allows for `sequence<octet>` to be accessed directly via a pointer to a memory buffer with variable size. This data type therefore perfectly fits the needs for direct deposit handling. In the following we use the octet streams as the base for the implementation of an optimized direct deposit ORB.

In MICO `sequence<octet>` is mapped to the C++-class `SequenceTmpl <Octet, MICO_TID_OCTET>` using the generic template `SequenceTmpl<>` which is used for all types of CORBA sequences. `SequenceTmpl<>` describes the records of a sequence by using the STL `vector<>`-type. This solution is very elegant and quite efficient for complex and heavily typed data structures. But for the handling of octet streams which are just streams of Bytes such processings lead to a large overhead.

5.5.4 Data Path Analysis

The data path of a static CORBA method invocation within the MICO ORB is depicted schematically in Figure 5.6.

Say that a client wants to communicate with a server. Therefore it allocates its data and passes it per reference through the 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 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 GIOP 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 `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 `StaticMethodDispatcher` which calls the compiler generated `object skeleton`. The skeleton maps the call to the requested server object method that implements the user functionality.

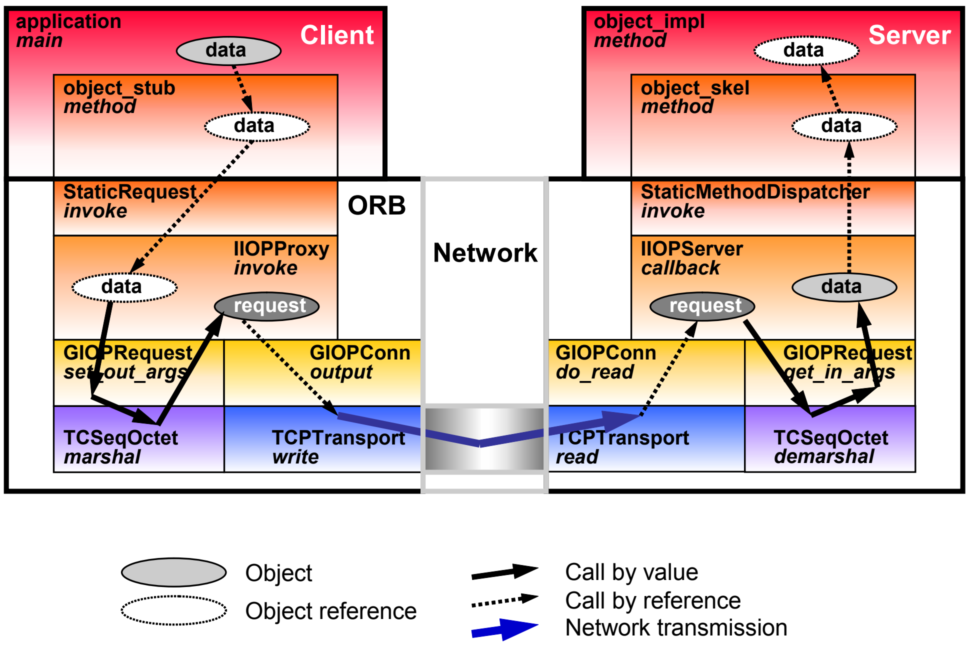


***Figure 5.6**: Data path through the MICO ORB.*

The problem that is introduced by the marshaling routine are the data copies (depicted by black arrows in Figure 5.6). The marshaling and demarshaling is handled by the `StaticTypeInfo` class. This class defines the virtual methods `marshal (DataEncoder &, StaticValueType)` and `demarshal(DataDecoder &, StaticValueType)`. For each of the CORBA parameter types there exist a concrete subclass of `StaticTypeInfo` that implements `marshal(DataEncoder &, StaticValueType)` and `demarshal(DataDecoder &,StaticValueType)`.

These subclasses of `StaticTypeInfo` are instantiated by the ORB on ORB startup. The appropriate subclass is selected by the TID of the data type that has to be marshaled or demarshaled respectively.

The marshaling and demarshaling routines in the `StaticTypeInfo` subclasses implement their functionality by taking the `StaticValueType` parameter data and copying it to a CORBA-request buffer allocated by the `DataEncoder` or the `DataDecoder` using an unoptimized loop and block wise `memcpy()`. The `IIOPProxy` then uses the `GIOPRequest` class to generate a GIOP-message of the type ’Request’. When the request message is ready in the `IIOPProxy` the next step is to initiate a connection which is done by the `GIOPConn` class and then send the message via the `TCPTransport` layer.

On the server the data is received by the `do_read` method of the server side `GIOPConn` class. This method is called by a callback function which is installed for all new connection. The method `do_read` runs until all the data for a GIOP-request message is received and stored in a buffer and then passed up by a further callback in the `IIOPServer`. The `IIOPServer` calls the demarshaling for the received GIOP-request and searches the specified object implementation in the `StaticMethodDispatcher`. Finally the unpacked parameters can be passed up to the requested method implemented by the application programmer. This method is wrapped by the server `skeleton` code which is a base class of the object implementation.

5.6 Modifications of MICO for Zero-Copy

5.6.1 A New Datatype: Sequence of ZC_OCTET

As already stated in Section 5.5 we focus on the CORBA-type `sequence<octet>` with its internal representation `SequenceTmpl <Octet, MICO_TID_OCTET>` as a basis for the implementation of a data type optimized for direct deposit. To compare an optimized stream version and parallelly allow the standard types we introduced a new type `ZC_Octet` which depicts a modified `Octet` type. The representation of this new type is identical to the standard `Octet`, both types are mapped to the C++-type `unsigned char`. Together with the introduction of a `MICO_TID_ZC_OCTET` as new type identifier it becomes possible to instantiate objects of the type `SequenceTmpl <ZC_Octet, MICO_TID_ZC_OCTET>`. The methods of this object type are further modified to support zero-copy direct deposit.

As the internal data representation of `SequenceTmpl<>` is an STL `vector<>` which is not suitable for direct deposit we had to find ways to store the data as untyped data directly in a memory buffer. We therefore extended the definition of the `SequenceTmpl<>` class by such a buffer. A pointer to a reserved memory block as well as a pointer that depicts the used page aligned area in this buffer and a long value that stores the effective buffer size was introduced in the class.

To implement the desired functionality of the direct deposit sequences we further added a case statement into the template class which allows to separate the direct deposit case from the standard case by using the template argument `TID`. This distinction of cases had to be implemented for different methods:

- `SequenceTmpl()`

  Standard Case: Empty

  Direct Deposit: Initialization of `buf`, `buf_aligned` and `size`

- `~SequenceTmpl()`

  Standard Case: Empty

  Direct Deposit: Deallocation of allocated memory in `buf` if not yet released.

- `length(MICO_ULong)`

  Standard Case: Setting of the length argument of the `vector`.

  Direct Deposit: Deallocation of allocated memory in `buf` and newly allocate a large enough buffer for `buf` with `malloc()`. Then the `buf_aligned` has to be set appropriately as well as the `size` argument. This is not optimal for the performance of such a call but can be modified in further optimization steps. Another aspect of this is that we expect the user to set the length of a sequence just once at the beginning when he allocates the memory.

- `length()`

  Standard Case: Return of the size of `vector`.

  Direct Deposit: Return of `size`.

- `operator[](MICO_ULong)`

  Standard Case: Return of the pointer to `vector[]`.

  Direct Deposit: Return of `buf_aligned`.

To allow an exchange of readily received data blocks between two `SequenceTmpl<>` without needing a copy we additionally introduced the following method:

- `copy_buffer(SequenceTmpl<> )`

  Standard Case: Empty

  Direct Deposit: Setting of `buf`, `buf_aligned` and `size` to the values of the passed parameter `SequenceTmpl<>`.

With this specializations of the `SequenceTmpl<>` methods we can enable MICO to internally handle direct deposit data. These methods on the other hand are exposed to the application developer too which enables to handle zero-copy data correctly in the application. What is especially useful is the `length(MICO_ULong)` method which is used for the initialization of a data block of a certain length and `operator[](MICO_ULong)` to access a data block of interest.

Finally the IDL compiler has to be modified to support the new `ZC_Octet` type. As discussed at the beginning we wanted to use both the standard and optimized octets parallelly why we introduced this new type. We therefore have to advice the IDL compiler to generate `ZC_Octet stubs` and `ZC_Octet skeletons`. These look the same as the standard sequence stubs and skeletons but just introduce the `ZC` prefix and the appropriate `MICO_TID`.

5.6.2 Direct Deposit Sender

As MICO chooses to statically instantiate methods for marshaling and demarshaling according to the `TID` of the used CORBA data type it was necessary for `SequenceTmpl <ZC_Octet, MICO_TID_ZC_OCTET>` to implement a concrete subclass `TCSeqZC-Octet` of `StaticTypeInfo`. This class provides the method `marshal( DataEncoder &, StaticValueType)` which is called by the `IIOPProxy` to generate a CORBA-request.

In the case of a direct deposit the data are not marshaled but just passed further to the TCPTransport layer (see Figure 5.7). On the other hand a GIOPRequest header is generated which contains the size of the data block to be sent 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` the data is sent to the `TCPTransport` layer directly by the marshaling routine `marshal (DataEncoder &, StaticValueType).`

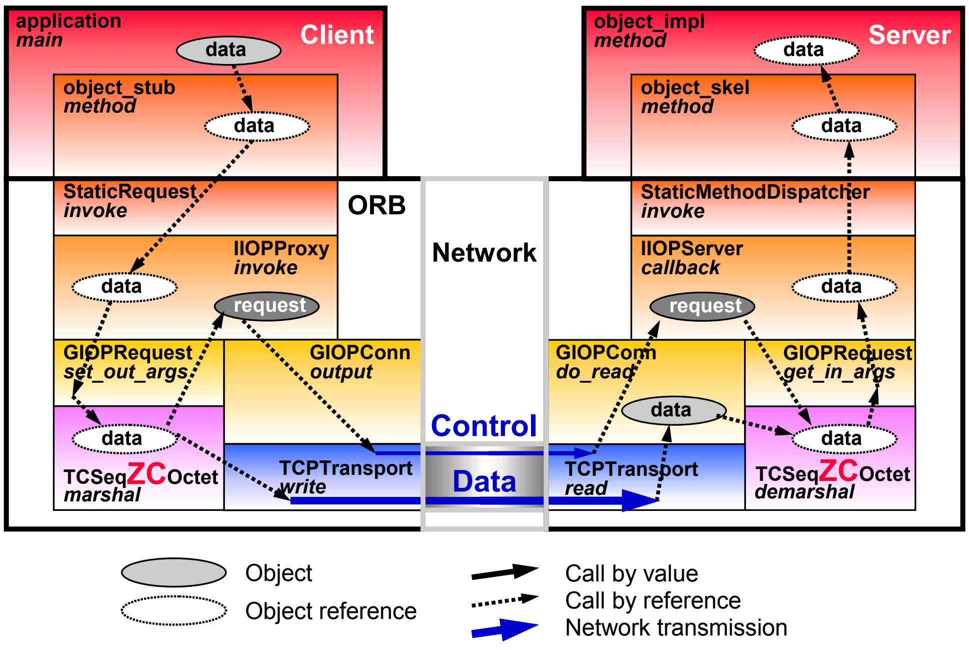


Figure 5.7: Data path through the optimized MICO ORB. The data copies are replaced by object references and a separation of control and data transfer is introduced.

To allow the receiver to allocate a large enough memory buffer for a direct deposit transfer even in the case of a delayed CORBA-request the marshaling routine sends an additional Zero-Copy header before sending the data (Figure 5.8. This header is page aligned and has the exact size of a memory page. The only data field that it contains is the size of the following zero-copy data block. An optional implementation optimization could combine the GIOPRequest header and the Zero-Copy header into this memory page.

As the Zero-Copy header is aligned and of the size of a memory page it synchronizes a speculative zero-copy receiver therefore preventing garbling of the first packet (see Section 4.3.7).

The Zero-Copy header is especially useful when using two dedicated connections over two different networks as described in Section 5.6.4. As this sort of headers are not supported by the IIOP protocol this type of extention implements an ESIOP, an Environment-Specific IOP as explained in more detail in Section 5.4.7.

5.6.3 Direct Deposit Receiver

The receiver side communication of MICO has a totally different structure than the sender side. It relies on callbacks why it was not possible to integrate the data reception directly in the `TCSeqZCOctet` demarshaling routine similar to the sender. Instead of using the demarshaling routine we took the `do_read()` method in the `GIOPConn` class and added a case statement that separates the normal case form a direct deposit receive callback.

In the case of a direct deposit request the initialization supplies a memory page buffer for the Zero-Copy 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 data block it 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 and `do_read()` terminates.

After the CORBA-request (without direct deposit data blocks) has been received by `do_read()` the demarshaling and method upcall is triggered via the `IIOPServer` by a further callback. While the rest of the CORBA-request is handled normally, the call of `demarshal (DataDecoder &, StaticValueType)` uses the new `copy_buffer (SequenceTmpl<> )` method for the direct deposit data. This allows to just set the pointers to the data in the data structures that are the passed to the method call. With this a passing per reference becomes feasible and copying is not used anymore.

5.6.4 Separate High-Bandwidth Connection

To prevent the garbling of messages as described in Section 4.3.7 from the outset we implemented a solution in MICO that uses two available networks concurrently for the communication (Figure 5.8).

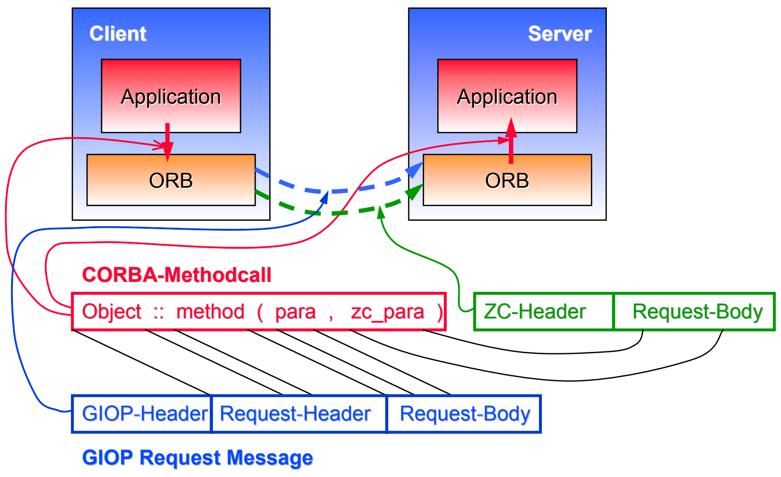


***Figure 5.8**: The optimized ORB uses separation of control and data transfer. Control transfer and standard GIOP messages are handled by a standard IIOP connection while zero-copy data is sent over a special zero-copy connection.*

The standard IIOP-connection is maintained as is. The connection is established for each request like in the unoptimized case and all the standard IIOP messages are further sent over this network. But for GIOPRequests which contain data containing direct deposit aware parameters this parameters are separated.

An additional gigabit-connection for direct deposit is established concurrently to the standard IIOP-connection for requests transferring zero-copy streams. For the TCP-Transport layer this just means that the *socket option* `SO_ZERO_COPY` is set to allow zero-copy data passing with the zero-copy sockets. This connection is established exclusively for method parameters specifying direct deposit behavior.

The fist connection that is established is always the standard IIOP-connection that just uses the IP-address specified by the object IOR. To find a second adapter we use the `ZC_Resolver` class which resolves the IP address of an available Gigabit Ethernet Adapter in the specified host. These addresses are then used to establish the additional gigabit-connection.

The helper class `ZC_Resolver` was introduced for the mapping between Fast- and Gigabit Ethernet adapters. The only method it exports is `Address get_zc_address (const Address)` which accepts an IP address as parameter and returns the IP address of the Gigabit Ethernet adapter of the specified machine.

This additional direct deposit connection is administered by the `IIOPProxy` and `IIOPServer` in the IIOP layer. MICO provides caching facilities that keep connections up to minimize the connection setup time. Gigabit connections profit from this caching too.

To anytime distinguish between different connections, standard and direct deposit, the list of by MICO supported protocols was enhanced. The protocols are `local` (local connection within an ORB for optimized interprocess communication), `unix` (for UNIX sockets), `inet` (for TCP/IP sockets) and `inet-dgram` (for UDP/IP sockets). This list was expanded by `inet-zc` depicting a zero-copy TCP/IP socket. By explicitly setting the protocol `inet-zc` to an address it becomes possible to force the utilization of direct deposit transfers as long as the adapter supports it.

5.7 Performance Evaluation

5.7.1 TTCP - TCP Performance Benchmark

The data for the experiments was produced and consumed by an extended version of the widely available TCP protocol benchmarking tool `TTCP` [154]. This tool measures the end-to-end data transfer throughput in MBit/s from a transmitter process to a remote receiver process. The flow of user data is unidirectional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters may be selected at run-time. These parameters include the number of data buffers transmitted, the size of data buffers, and the size of the socket transmit and receive queues.

The following versions of `TTCP` were implemented and benchmarked:

Raw TCP version This is the standard TTCP program implemented in C. It uses C 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 Chapter 4. This is more or less done by specifying a zero-copy socket option and data buffer aligning.

CORBA version This version replaces all C socket calls in TTCP with stubs and skeletons generated from a pair of CORBA IDL specifications. 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. As the zero-copy implementation provides its optimization to 4 KByte pages only, the data buffers were run in 4 KByte increments. Each test was run at least 10 times to account for performance variation due to transient load on the networks and hosts. The variance between runs was very low since the tests were conducted on otherwise unused networks.

5.7.2 CORBA Implementation of TTCP

The CORBA implementations were developed using a single threaded version of MICO 2.3. Extending `TTCP` to use CORBA required several modifications to the original C-socket code. All socket calls were replaced with stubs and skeletons generated from a pair of CORBA interface definitions. The IDL interface uses a sequence of octets to transmit the data as follows:

```
typedef sequence<octet> Buffer;

interface ttcp
{
    void send(in Buffer data);
    void send_inout(inout Buffer data);
    oneway void send_oneway(in Buffer data);

    void prep_timer();
    void do_stats();
};
```

The data buffers exchanged between the sender and receiver in TTCP are treated as a stream of untyped bytes. The send operation uses oneway semantics since the TTCP benchmarks measure the performance of uni-directional data transfer. This behavior is consistent with the flow of communication in electronic medical imaging applications and video distribution. Additionally a ping-pong test as well as a standard twoway function which block until the function returns was implemented.

The client side of the original TTCP was modified to bind to a TTCP server by an object reference to the object implementation of TTCP. Once the object references were obtained, data buffers of the appropriate size were initialized and then transmitted by calling the IDL-generated send stubs.

The server-side was modified to create an object implementation for TTCP. CORBA IDL compilers generate skeletons that translate IDL interface definitions into C++ base classes. Each IDL operation is mapped to a corresponding C++ pure virtual method. Programmers then define C++ derived classes that override these virtual methods to implement application specific functionality.

5.7.3 Hardware Platform

The experiments in this section were conducted using our first generation CoPs cluster. The machines are identical to that of Chapter 4, we use off-the-shelf 400 MHz Pentium II PCs, running Linux 2.2, connected via Gigabit Ethernet by fiber optic cables. Our Gigabit Ethernet test bed comprises a SmartSwitch 8600 manufactured by Cabletron and GNIC-II Gigabit Ethernet interface cards manufactured by PacketEngines. For parts of the test we use our optimized de-/fragmenting driver with speculation. This allows us to test real zero-copy TCP communication with middleware across all layers.

5.7.4 Experimental Setup

Figure 5.9 shows the network configuration we use in our cluster. Each machine hosts three Ethernet adapters. The `eth0` is the system and maintenance network that is used for NFS access, login and remote machine control. A separate Ethernet was used for the measurement of the IIOP-requests and the CORBA method invocations. The direct deposit transfers are send over a switched Gigabit Ethernet network. The machine is further described in Section 5.7.3.

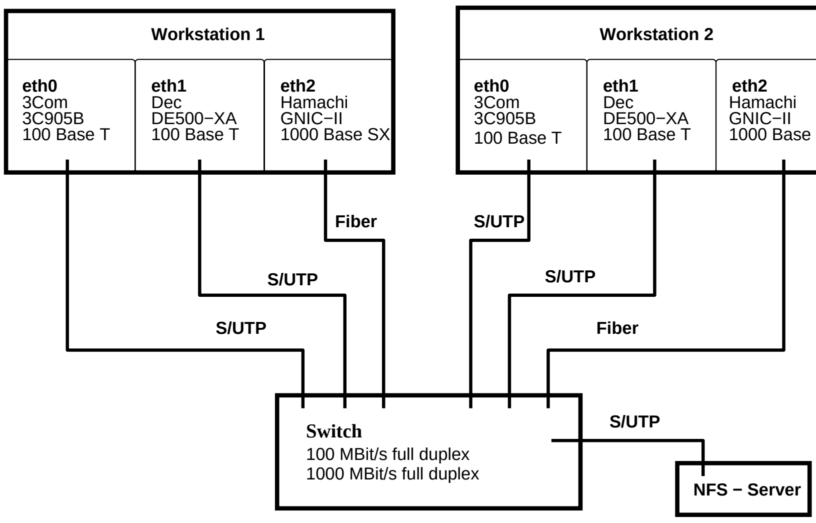

Workstation 1
eth0 3Com 3C905B 100 Base T
eth1 Dec DE500−XA 100 Base T
eth2 Hamachi GNIC−II 1000 Base SX
Workstation 2
eth0 3Com 3C905B 100 Base T
eth1 Dec DE500−XA 100 Base T
eth2 Hamachi GNIC−II 1000 Base SX
Fiber
S/UTP
S/UTP
S/UTP
S/UTP
S/UTP
Fiber
S/UTP
Switch
100 MBit/s full duplex
1000 MBit/s full duplex
NFS − Server

***Figure 5.9**: As our measurement platform we used machines equipped with three networking interfaces: a service network, a separate Fast Ethernet network used for IIOP transfers as well as a Gigabit Ethernet network for zero-copy transfers.*

For the measurement of the direct deposit support we used the original Linux 2.2 TCP/IP stack and the standard network card drivers. The measurements using zero-copy TCP we run our zero-copy optimized stack with the speculatively defragmenting hamachi driver. The test application were the three version of TTCP described in Section 5.7.1.

The CORBA based measurements were done with the CORBA TTCP and the optimized ORB as discussed. For the measurements that shall not profit from the optimizations the interface just specified the standard `sequence<octet>` type instead of the optimized `sequence<ZC_octet>`.

Again 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. As the zero-copy implementation provides its optimization to 4 KByte pages only, the data buffers were run in 4 KByte increments. Each test was run at least 10 times to account for performance variation due to transient load on the networks and hosts. The variance between runs was very low since the tests were conducted on otherwise unused networks.

5.7.5 Performance Results of MICO With Zero-Copy

Figure 5.10 illustrates the results of measuring two versions of the `TTCP` benchmark implemented on top of two different versions of CORBA.

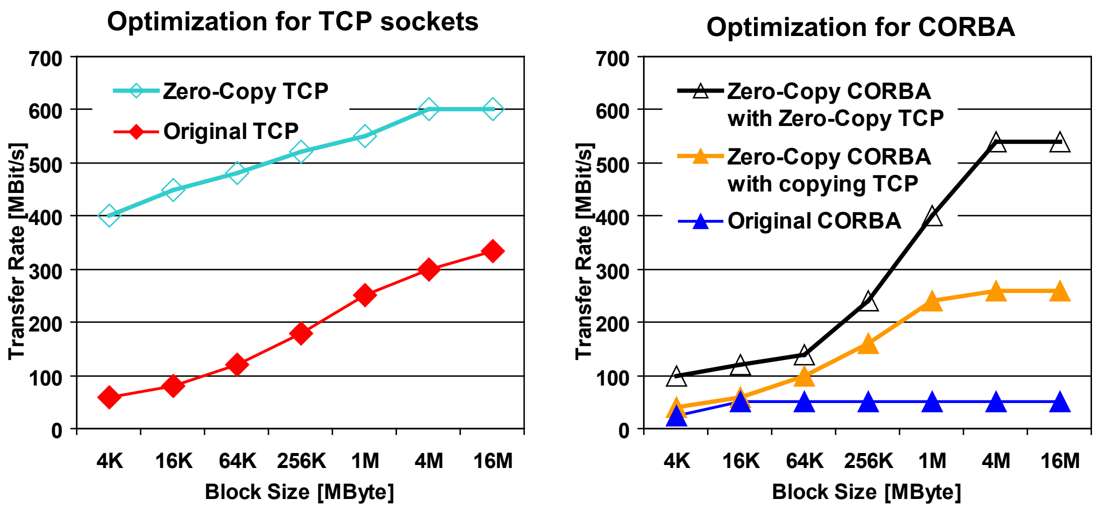


***Figure 5.10**: Bandwidths measured with TTCP for raw TCP-sockets and for CORBA. The left chart shows the performance gain for the zero-copy socket interface, the right chart the gain of the zero-copy optimization in the MICO ORB. The optimized ORB is able to sustain the same bandwidths as the simple TCP-socket benchmark.*

A first point that is shown nicely again in the chart is that the zero-copy TCP stack performs much better then the original copying stack. The large performance gain for small messages are achieved by the optimized latency and the small read and write overhead which allows to achieve very good throughput already by transmitting single pages.

But the contribution of this chapter is shown in the ’Optimized CORBA’ measurement which nicely matches the C-socket version of TTCP. That proves that the optimized ORB handles the `ZC_Octets` correctly by just passing it through the ORB while not introducing much overhead.

A third aspect is shown in the ’Zero-Copy CORBA’ measurement which uses both the optimized CORBA ORB and the zero-copy TCP/IP stack. For large blocks our ORB achieves 550 MBit/s throughput while the application still fully complies tho the CORBA standard.

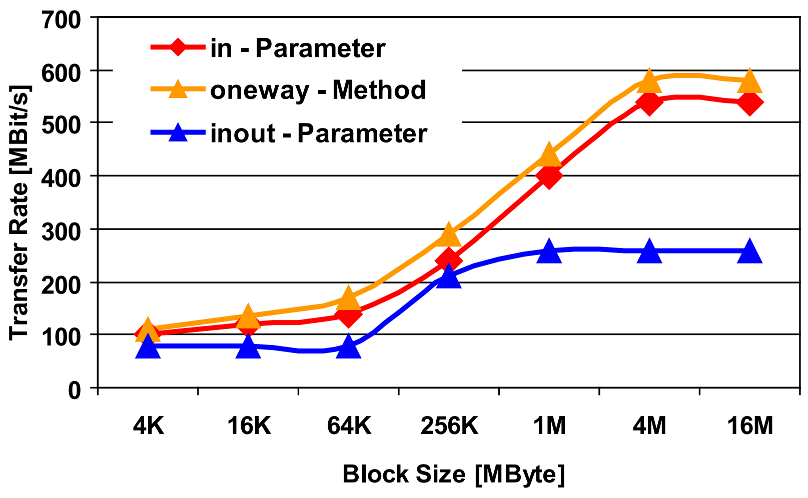


***Figure 5.11**: CORBA direct deposit transfers using different parameter types.*

A further series of measurements shown in Figure 5.11 examines the different parameter and method types `in`, `inout` and `oneway`.

in parameter Standard twoway CORBA-method declaration with parameter type `in`.

oneway method CORBA-`oneway`-method declaration with parameter type `in`.

inout parameter Standard twoway CORBA-method declaration with parameter type `inout`.

Until now we used the CORBA standard twoway to do method invocations meaning that a two way communication takes place and a call is blocking until the server responds. The data buffers exchanged between the sender and receiver in TTCP are treated as a stream of untyped bytes and the send operation uses a oneway semantics. Since the TTCP benchmarks measure the performance of uni-directional data transfer we test oneway semantics with CORBA too. This is possible for methods that do not return any parameter or function value which is consistent to the behavior of the flow of communication in electronic medical imaging applications and video distribution. Even if the oneway method is not 100% reliable in the sense that the client does not know if the server accepted a call and the called method is available the communication of the GIOPRequest takes place over a reliable TCP transport layer meaning that no data gets lost on the way to the server. Additionally a ping-pong test which blocks until the function returns was implemented.

The result shows a slightly better performance for the `oneway` methods which is clear as the overhead of the acknowledgment is ignored. But the curves still look very similar as the overhead for large blocks can nearly be neglected. But still the oneway method can be used for non blocking calls which does not further improve the network performance but can improve the overall performance of an application as the computation can further proceed.

The twoway ping-pong method using `inout` parameters achieves 50% of the performance as expected because the resulting bandwidth just considers one transfer. As the same amount of data is transfered back, the result could be doubled to get the bandwidth that is achieved on the network. For smaller packets it makes sense that this doubled bandwidth is slightly better than what can be achieved by a uni-directional transfer.

5.7.6 Benefit of Zero-Copy Middleware

The performance results achieved with zero-copy CORBA look very promising. It was the goal of this chapter to optimize a CORBA ORB for direct deposit operation. As shown in the results our optimized MICO-ORB nicely matches the C-socket performance. That proves that the optimized ORB handles the `ZC_Octets` correctly by just passing it through the ORB while not introducing much overhead.

It is astonishing that the optimized ORB can keep up with the much improved performance when using the zero-copy TCP stack. This is very encouraging and proves that our argumentation about zero-copy as the most essential technique to optimize system software for efficiency is true.

The results described in Section 5.7.5 show the success of separating control and data transfers in the ORB. The higher level conceptional contribution that was the intent of this chapter shows that the concepts really work. It was namely possible to prove that it is indeed possible to use CORBA middleware on top of a standard socket application programming interface and still adhere to the zero-copy regime. For large blocks our ORB achieves 550 MBit/s throughput while the application still fully complies tho the CORBA standard. That means that the programmer just defines the interface and generates the stubs and skeletons which map the server implementation. This stands in relation to the 50 MBit/s that are achieved by a copying ORB and the standard TCP/IP stack. This means that a performance gain of 1000% could have been achieved by introducing zero-copy.

Of course the separation of control and data transfer introduces some latency for small amounts of parameter data. We therefore loose some of the benefit. But still a problem for the effective transfer of small parameter sizes is the latency imported by using the ORB itself that was not much improved in our implementation. These overheads are introduced by interpreting the object reference, connection establishment as well as IIOP related overhead. They largely dominate so that the effective size of the small parameters did not show any difference in time for a method call. But as we are focusing on high bandwidth networks transporting huge amounts of data this is negligible. And still the optimized ORB is much faster than the original MICO.

6

Modeling Distributed Objects for Parallel Processing

A growing interest in distributed technologies for high performance distributed computing is driven primarily by the popularization of the Internet. This results in new challenges for software developers of distributed applications that must deal with complex issues, such as remote communication, partial failures and concurrency management.

With the advent of clusters, the focus of system architecture in Parallel- and Distributed Computing has shifted away from unique massively parallel supercomputers towards smaller and more cost effective systems built entirely with personal computer parts. This trend leads to the overwhelming popularity of Clusters of PCs to run traditional supercomputer application codes. A closer look at this trend reveals a continuing evolution towards even more parallelism and towards a wider distribution e.g. on grid computing. The better connectivity of most computers on the Internet will shortly permit a new view of things suggesting that all those machines form a giant global computational grid.

This vision of a grid requires a set of new tools to program such systems. In the last few years middleware has emerged as an important architecture for supporting rapid development of distributed applications. Several programming environments that greatly reduce the complexity of developing distributed software have been proposed and used. These environments provide high-level facilities for developing distributed applications without having to deal with low-level details, such as remote communication and object location. They use object-oriented abstractions and software components to deal with the complexity of the system and promote modularity and reusability. CORBA [112] is an example of an object oriented environment supporting simple distributed objects and components as well.

These environments however lack a general support for “one-to-many” communication primitives; such primitives greatly simplify the development of several types of high performance applications that have requirements for parallel processing, collaborative work, high availability, or fault tolerance.

There exist more traditional software systems for parallel programming (e.g. APIs and middleware) as candidates for high performance distributed computing and they use the same distributed computing platform technology as CORBA: multiple interconnected computers cooperating on an application. But such systems have normally been developed in a way that remained disconnected from object-based and distributed object computing e.g. with simple Message Passing Interface (MPI) [57] or Parallel Virtual Machine (PVM) [147].

Lacking support for objects prevents parallel applications from taking advantage of the large investment in distributed object software technologies and tools, and also makes it difficult to add parallel parts to distributed CORBA applications. Therefore parallel programmers still require a lot of redundant learning about significantly overlapping technologies (e.g. typed message passing [58]). Similarly, CORBA applications that need parallel subsystems require the CORBA-programmers to learn about low level message passing for no good technical reason.

This chapter proposes some ideas on how to model distributed objects for parallel processing. The optimized MICO ORB shall then be used to parallelize jobs and distribute data without internal copies resulting in an efficient implementation that matches simple message passing in performance. We observed that none of the current approaches is able to completely satisfy the requirements given by the MPEG-Transcoder application described in Chapter 7 and therefore introduce the design of a new service which we named simply a *CORBA Distributor Service*. This type of service matches the application pattern we aim at and allows transparent parallelization with decoupled clients and servers. With an inherent load balancing service the CORBA Distributor Service serves as the basic foundation to implement parallel applications on clusters of PCs with object and component based technology that still adheres to the zero-copy principle by just using an optimized ORB.

6.1 Parallelization with CORBA

Parallel applications are characterized by a set of processes operating in parallel, usually on parts of a larger data set that is divided up among the participating processes. Data is typically redistributed between a set of sending processes and a set of receiving processes, which are often the same single set.

This pattern has been implemented in several CORBA-based environments by modeling the data distribution as aggregate data arguments to a CORBA invocation. These arguments are then divided up, collected, or redistributed between one or more clients and one or more servers.

CORBA implements an inherent synchronous client/server paradigm (N:1) which has generally been considered unsuitable for parallel programming due to the lack of peer-to-peer semantics and difficulty in achieving distributed concurrency and/or data flow. So what is needed is a parallelization paradigm (1:M) (see Figure 6.1).

A number of these issues have been mitigated by recent evolutions of CORBA such

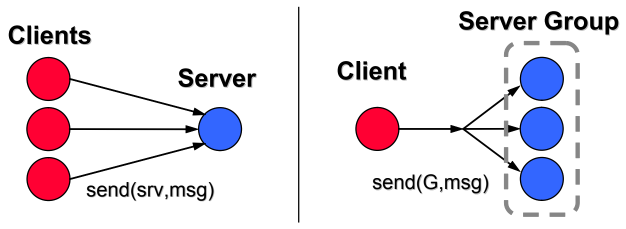


***Figure 6.1**: CORBA implements an inherent synchronous client/server paradigm (N:1). But for parallelization a (1:M) paradigm is needed.*

as asynchronous method invocation [107] which is now part of the new CORBA 3 [114], multithreading, and reactive/recursive ORB implementations (which can process a request while waiting for a reply, all in a single thread). However, not all interactions can be described in today's CORBA model.

Several independent projects have demonstrated CORBA's usefulness for parallel programming generally by extending the interactions available and supporting some level of data partitioning. This has typically resulted in some non-standard ORB extensions that are neither portable nor interoperable. We intended to go another way and stay within the standard CORBA specification for the ORB adding all required functionality to a new CORBA service. The CORBA standardization process by the Object Management Group (OMG) can then pick up our ideas and incorporate them into a new CORBA Common Object Service (COS).

6.2 Reusing CORBA Patterns for Parallel Processing

When adopting a service approach, one might wonder whether an existing CORBA service does already provide a suitable paradigm for object replication or call forwarding and parallelization, or in more general terms if some IDL interfaces of an existing service could be reused. Our evaluation of existing CORBA services as well as CORBA ORB extensions has shown, that none of the services and standard extensions is able to meet all the requirements to implement the functionality we were focusing on. Nevertheless some of the COS services contain certain techniques that are interesting to be considered for parallel processing. This following section of my thesis looks into some existing services and into the criteria that are met or missed by previous solutions.

6.2.1 COS Event Service

A CORBA method call to a server object generates a synchronous request. The caller passes its parameters and then blocks until the request has been processed and a return value is returned. But there are scenarios that need a decoupled communication.

The COS Event Service [110] decouples the communication between suppliers and consumers through so called *Event Channels*. Suppliers produce event data and consumers process event data. Suppliers can generate events without knowing the identity of the consumers. Conversely, consumers can receive events without knowing the identity of the suppliers. This scenario is also known as the Publish&Subscribe paradigm.

An event channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events, and can thus produce or consume events from another event channel. Additionally the channel manages the registered suppliers and consumers.

The COS Event Service provides “one-to-many” communication that can be used potentially for parallelization purposes. But the event service does not provide return values. Two way communication can be achieved through reverse channels if needed.

Event data is communicated between suppliers and consumers by issuing standard CORBA requests. Events by themselves are not CORBA objects because the CORBA object model at the time of the specification of the COS Event Service did not support passing objects-by-value. There are two approaches to initiating event communication between suppliers and consumers. These two approaches are called the push model and the pull model.

The push model allows a supplier of events to initiate the transfer of event data to consumers. The pull model allows a consumer of events to request event data from a supplier. In the push model, the supplier is taking the initiative; in the pull model, the consumer is taking the initiative.

Event channels are standard CORBA objects that allow consumers and suppliers to exchange events. Communication with an event channel is accomplished using standard CORBA requests. An example configuration of the event service is illustrated in Figure 6.2.

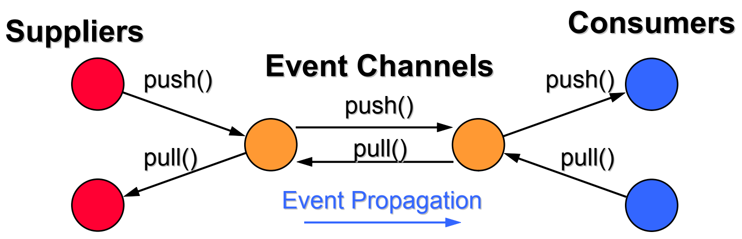


***Figure 6.2**: Event Service Example Configuration*

The basic event service model only allows the sending of untyped data which is written as of type ’any’. Hence, it provides a message-passing-like interface to event communication. The COS Event Service also provides a typed model that offers RPC-like communication. Suppliers call operations on consumers (typed push model) or consumers call operations on suppliers (typed pull model) using operations of an application-specific IDL interface.

Replication with Event Channels

An event service provides a natural way for multicasting requests to replicated objects, using the push model with one event channel: all the copies of a replicated object are consumers of the channel, while clients supply event data on this channel (Figure 6.3).

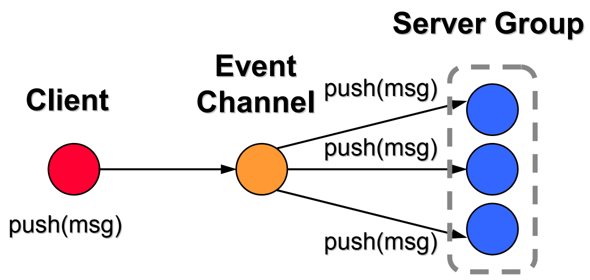


Figure 6.3: *Event Service for replication*

In particular, the typed version of the event service provides for straightforward transparent server replication: clients register with the event channel as suppliers and obtain a reference to an object that supports the same interface as the replicated server; they can then issue invocations on this interface directly. Clients do not need to know the number or location of the copies, which can change at runtime.

Limitations of the Event Channel Approach

The COS Event Service achieves a decoupling of client and server through asynchronous method calls. A referential decoupling is also achieved through the mediator role of a channel.

Both the referential decoupling as well as the asynchronous method calls are requirements for parallelization. However the COS Event Service just provides a "best effort, one-to-many" distribution semantic. But we need a decoupled "best effort, one-to-one" delivery. Another problem is that the event service lacks load balancing functionality since it is not needed in the domain of its normal operation.

Even using event channels for replication has some major limitations. There are no group management facilities, nor guarantees concerning ordering, atomicity, or failures. For instance, strict atomic delivery between all suppliers and all consumers would require additional interfaces. Different qualities of service may be provided by different implementations, but their alternative use and exchange ability are not standardized in the event service.

In addition, the model of the event service is not ideal for object replication. The event service only supports one-way communication, i.e., operations on the replicated server must have only in parameters. This restriction is very limited for many distributed applications that require results from invocations. Return values may be transmitted using a “reverse” event channel, but this requires clients and servers to be both consumers and suppliers (see Figure 6.4).

Another fundamental problem with the design of the event service is that it is centralized. Although consumers and suppliers use different interfaces for pushing and pulling event data to and from the channel, they have to invoke the same centralized object in order to connect to an event channel. This event channel is a standard CORBA object, which is a single point of failure in the event service architecture. There are several ways for decentralizing an event service [52]. The solution that we consider the most promising is also used in [53, 54] and consists in chaining event channels.

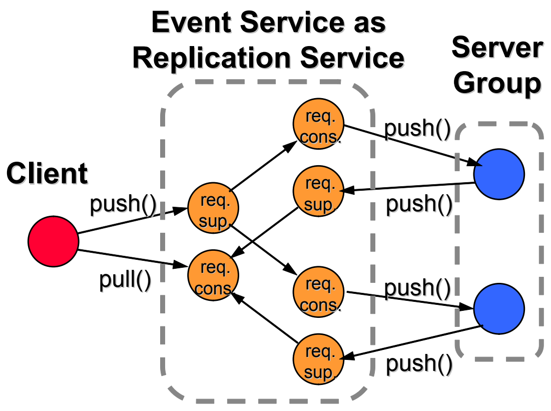


***Figure 6.4**: Two way communication via a reverse channel and chained Event Channels used for decentralizing.*

Rather than having a single event channel that diffuses messages to all copies of the replicated object, one could introduce several event channels located on the client and on the server site. A client is represented as a request-supplier and a response-consumer, while a server is represented by a request-consumer and a response-supplier (Figure 6.4). This model provides two-way communication with no single point of failure. Distinct clients generate data using distinct request-suppliers and receive replies through distinct response-consumers.

The request-supplier object performs multicast communication, possibly executing some protocol with the request-consumers for guaranteeing atomicity and ordering of messages. The response-consumer gathers multiple replies and returns them to the client through a push or a pull mechanism.

This approach is fully CORBA compliant and does not modify the COS Event Service specification. Also it does not introduce a single point of failure. Extra protocols are however necessary between the channel objects for ensuring atomicity and ordering of messages, and also for group membership.

6.2.2 COS Notification Service

The COS Notification Service [111] extends the existing COS Event Service and preserves all of its semantics allowing for interoperability between basic event service clients and notification service clients.

Countless projects had to stop using the basic standard COS Event Service because the service lacked certain features which then had to be integrated in a proprietary way killing any benefit of using a standard service.

Filtering and Quality of Service

There are two serious limitations of the event channel defined by the COS Event Service: (1) it supports no event filtering capability, and (2) it has no ability to be configured to support different qualities of service. A primary goal of the COS Notification Service is to enhance the COS Event Service by introducing the concepts of filtering, and configurability according to various quality of service requirements. Clients of the notification service can subscribe to specific events of interest by associating filter objects with the proxies through which the clients communicate with event channels. These filter objects encapsulate constraints which specify the events the consumer is interested in receiving, enabling the channel to only deliver events to consumers which have expressed interest in receiving them. Furthermore, the notification service enables each channel, each connection, and each message to be configured to support the desired quality of service with respect to delivery guarantee, event aging characteristics, and event prioritization.

In addition to these extentions the CORBA Notification Service additionally supports event filtering on three fundamental types of events: *untyped events* contained within a CORBA Any, *typed events* as defined by the COS Event Service, and *structured events*, which are introduced in this specification. Structured events define a well-known data structure which many different types of events can be mapped into in order to support highly optimized event filtering.

Limitations of the Notification Approach

The COS Notification Service like the COS Event Service supports asynchronous exchange of event messages between clients and a referential supplier/consumer decoupling. For the implementation of our solution the extensions of the notification service over the event service do not bring any advantages. Even if the notification service provides many kinds of quality settings the “one-to-many” delivery strategy remains the same. Therefore also the notification service does not contain a load balancer.

6.2.3 Object Groups in Fault Tolerant CORBA

While CORBA simplifies the distribution of objects the standard environment lacks support for *one-to-many-* and *many-to-many-*communication primitives. Such primitives can greatly simplify the development of several types of applications that have special requirements for data parallelism.

One-to-many interactions can be provided by group communication. This allows to manage *Groups of Objects* and provides primitives for sending messages to all members of a group, with various reliability and ordering guarantees. A group constitutes a logical addressing facility: messages can be issued to a group without having to know the number, identity, or location of individual members. The notion of group has proven to be very useful for providing high availability through replication: a set of replicas constitutes a group, but are viewed by clients as a single entity in the system.

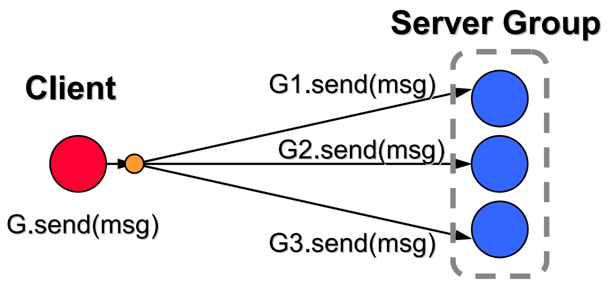


***Figure 6.5**: Group Communication used in Fault Tolerant CORBA*

The idea of adding group communication in an object-oriented middleware is not new. When we started this work, at least two products (Electra [99, 100, 101] and Orbix+Isis [81]) were available to support group communication in a CORBA-based environment. And the OMG was already working on a Fault Tolerant CORBA specification which has meanwhile been published as a standard in CORBA 2.5 and was integrated without major changes into CORBA 3 [115].

The standard for Fault Tolerant CORBA aims at providing robust support for applications that require a high level of reliability, including applications that require more reliability than can be provided by a single backup server.

Fault tolerance depends on entity redundancy, fault detection, and recovery. CORBA provides fault tolerance by the replication of objects. This strategy allows great flexibility in configuration management of the number of replicas, and of their assignment to different hosts, compared to server replication. Replicated objects can invoke the methods of other replicated objects without regard to the physical location of those objects.

Replication and Object Groups

To render an object fault-tolerant, several replicas of the object are created and managed as an object group. While each individual replica of an object has its own object reference, an additional interoperable object group reference (IOGR) is introduced for the object group as a whole. It is this object group reference that the replicated server publishes for use by the client objects. The client objects invoke methods on the server object group, and the members of the server object group execute the methods and return their responses to the clients, just like a conventional object. Because of the object group abstraction, the client objects are not aware that the server objects are replicated (replication transparency) and are not aware of faults in the server replicas or of recovery from faults (failure transparency).

Limitations of the Object Group Approach

As the replication relies on object groups that are depicted by an additional interoperable object group reference (IOGR) an unreplicated client hosted by a legacy ORB can in fact invoke methods of a replicated server, supported by the Fault Tolerance Infrastructure. And also the object group references generated for replicated servers can be used by legacy ORBs, although the full benefits of fault-tolerant operation are not achieved for an unreplicated client.

But the interoperability is worse within a fault tolerant domain where all of the hosts must use ORBs and fault tolerance infrastructures provided by the same vendor to ensure interoperability and full fault tolerance within that domain. This defies the CORBA concept. Consequently, the members of an object group must be hosted by ORBs from the same vendor and Fault Tolerance Infrastructures from the same vendor.

However object groups are a very interesting concepts but not generally available at this time yet. A change of the ORB would make it non-interoperable with other ORBs. And again also the object groups within Fault Tolerant CORBA provide a “one-to-many” distribution semantic while we need a decoupled “one-to-one” delivery for high performance distributed computing.

6.2.4 Load Balancing Service

Until now, there has been no COS Load Balancing Service specified by the OMG. But there is some native support in GIOP to support the re-direction necessary for load balancing in a robust and standard way. A special exception status (LocationForward), and the LocateReply message, are the specific features that load-balanced ORBs can use. They enable an ORB vendor to establish one or several machines as ”dispatchers”that receive a client’s initial request, designate a lightly-loaded machine as a proper host for the requested instance, and re-direct the client’s invocations to it (Figure 6.6). This load balancing mechanism is standard, and works even on cross-vendor invocations.

CORBA, however, does not yet standardize interfaces that allow an object or server implementation to work with the ORB on load balancing. An RFP issued by the OMG in 2001 will standardize these interfaces. This RFP seeks CORBA technology standardization for load balancing and monitoring support for CORBA-based applications that operate on computing servers, server-farms (web), computing clusters, and high-performance computing environments [76].

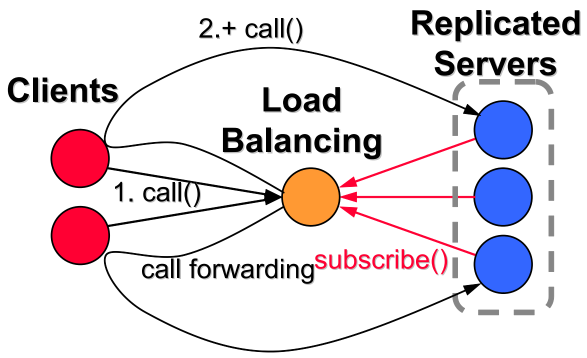


***Figure 6.6**: Load Balancing Service serving as a request dispatcher.*

The structure of the requirements for replica management - an essential feature of a COS Load Balancing Service - is similar to those proposed in Fault Tolerant CORBA [115] (and recently Data Parallel CORBA [113]). Several objects implementing the same interface can be merged into a load balancing group. The canonical notion of such a load balancing group module is just a subset of the IDL from the Fault Tolerant CORBA specification.

Call Forwarding

A client sends a request to the group and the request is transparently forwarded to a server with enough capacity to process the call. This is done by the “LocationForward” exception as discussed above. The request arriving to the group is therefore not processed but just send back with a hint to an other object. The client ORB again sends the request but now to the object reference that it got from the group.

Limitations of the Load Balancing Call Forwarding Approach

Unfortunately the forwarding mechanism is not particularly suitable for all load balancing aspects as most ORB implementations cache the “LocationForward” hints and future requests are directly send to the forwarding address. This is not the idea of a load balancing service we require but makes sense to many applications that access the same data several times or that rely on side effects like state information (sessions).

A COS Load Balancing Service also must specify mechanisms to tracking the load of its group members. This information is indispensable for an adaptive load balancing.

So the load balancing service implements the balancing functionality which was missing in the COS Event- and the COS Notification Service. It also supports dynamic subscribing and unsubscribing similar to event channels but unfortunately lacks the decoupling of client and servers.

6.2.5 Data Parallel CORBA

The Data Parallel CORBA specification [113] enables the same basic patterns of computation and communication manifested by high performance, scalable, parallel applications and systems, but under a CORBA-based programming model.

Adding Parallel Objects

The concept of *Parallel Objects* is an additional approach for the implementation and use of CORBA objects that enables the object implementer to take advantage of parallel computing resources to achieve scalable, high performance.

Parallel objects (Figure 6.7) are objects whose requests may be carried out by one or more singular objects, so called parts, probably, but not necessarily, running concurrently in different execution contexts. Thus, the work to process a request to a parallel object is carried out, in parallel, in multiple execution contexts. The implementation is such that different aspects of the work on a single request may be done piece by piece in parallel.

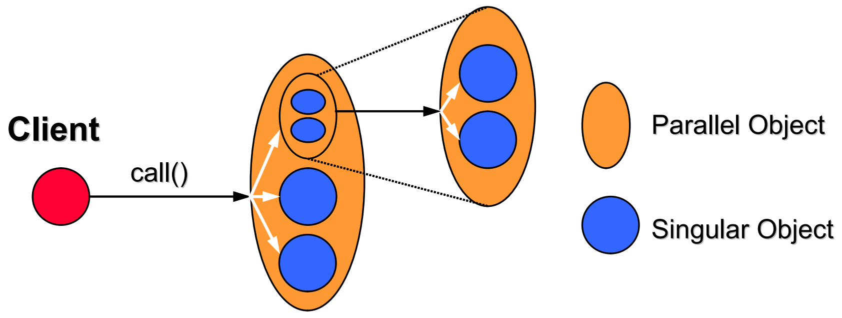


Figure 6.7: *Parallel Objects specified in Data Parallel CORBA. A parallel object designates a group of several singular server objects.*

The specification enables clients to use such objects transparently and efficiently, whether or not the client ORB supports the new features. This is again similar in spirit and structure to Fault Tolerant CORBA [115], which enables a replicated implementation alternative to achieve higher availability, which is also transparent to clients.

Parallel objects embrace many of the concepts and techniques embodied in other course-grained parallel programming APIs and systems. They are somewhat analogous to process sets or process groups in these other systems. Parallel objects, whose implementation is a set of partial implementations executing in parallel, can be used by normal CORBA clients, and can also make requests of normal CORBA objects. They can also been cascaded in the sense that they can do requests on other parallel objects and also on themselves.

Scalability of parallel objects is a run-time issue enabling reusability of scalable parallel implementations. The specification requires a parallel-capable ORB for running a client to participate in making invocations on parallel objects.

Data Partitioning

The Data Parallel CORBA specification also defines interfaces and semantics for the partitioning and distribution of the data and requests involved in the use of parallel objects. Since the implementation of parallel objects is generally distributed in a homogeneous pattern across a set of parallel computing resources, this capability supports parallel implementations. The used techniques embrace many of the concepts specifically defined in the Data Reorganization Effort that has collected and consolidated best practices in this area.

Limitations of the Parallel Objects Approach

A first limitation applies to parallel objects as it does to Fault Tolerant CORBA. They both rely on object groups with dedicated object group references (IOGR). This needs a redesign of the ORB and makes the system non-interoperable with other ORBs.

As a second limitation Data Parallel CORBA is restricted to using the parallel computing resources in homogeneous, data parallel patterns, rather than some arbitrary forms of work decomposition. This is similar to jobs that are parallelized by a parallelizing compiler like for High Performance Fortran. Such compilers e.g. divide loops into parts and redistribute the data onto different processors to process these independent part-loops.

In contrast to this pattern we aim at a more general form of work decomposition. Another point is that our data might not be available on its whole at the moment when the first tasks can be started. Say a video processing application as designed in Chapter 7 continuously gets data to process from a frame grabber. In such a scenario it is not possible to call a method with the whole chunk of data that is afterwards partitioned and parallelly processed. We rely on decoupled parallel processes that can be started at anytime, if possible in parallel.

6.3 Ways to Integrate Parallelism into CORBA

The CORBA object model defines an object as an entity with a well-defined interface that may be remotely invoked using an object reference. An object reference is an "object name that reliably denotes a particular object". This means that the CORBA specification without object group references (IOGR) does not permit an object reference to designate a set of objects, and it does not provide ways for clients to invoke several objects at once using an object reference. CORBA only deals with point-to-point remote invocations.

The absence of mechanism that permits the decoupled requests to groups of replicated CORBA server objects complicates the design and implementation of many applications that have requirements for parallel processing. During the last couple of years, several systems have been developed to augment CORBA with groups which is somehow the base technique to handle replicated objects. [51] classifies these systems according to three main categories, each of which represents a different approach to group communication in CORBA:

1. The **integration approach** integrates an existing group communication system within an ORB.

2. The **interception approach** intercepts messages issued by an ORB and maps them to a group communication toolkit.

3. The **service approach** provides group communication as a CORBA service beside the ORB, and was chosen as the basis of this thesis.

We now describe these three approaches. We also discuss a variant of the service approach, which consists in adapting the interfaces of an existing CORBA service – the event service – rather than defining new interfaces for group communication.

6.3.1 Integration Approach

With the integration approach, the ORB functionality is enhanced by a group communication toolkit. The ORB directly deals with object groups and references to object groups. CORBA requests are passed to the group communication toolkit that multicasts them from clients to replicated servers, using proprietary mechanisms. The group toolkit is "integrated" into the ORB.

Existing systems that use the integration approach (e.g. Pardis [86] but also prototype implementation of Fault Tolerant CORBA). The basic idea is to extend the IDL language mapping and to generate two types of functions from IDL definitions: (1) standard functions that conform to the language mapping and (2) special functions with sequences of values for out and inout parameters. The client uses the function with the signature that corresponds to its needs. If the client is not aware of groups, it uses only standard functions.

6.3.2 Interception Approach

With the interception approach, the ORB is not aware of replication. We have already discussed in Section 5.1.2 how extensions for high performance can be incorporated into an ORB. ORB requests formatted according to the IIOP protocol can be intercepted transparently on client and server sides using low-level interception mechanisms provided by CORBA Interceptors; they are then passed to a group communication toolkit that forwards them to the parallel processes. This approach does not require any modification to the ORB, but relies on interception of requests and on a further toolkit implementing the parallelizing functionality.

6.3.3 Service Approach

The service approach, which has been adopted and developed in the context of this chapter, can provide explicit group and support for parallelism through a CORBA service. Unlike the integration approach, a CORBA service is mostly specified in terms of IDL interfaces, and does not depend on implementation language constructs. The ORB is not aware of groups, and the service can be used with any compliant CORBA implementation. The service approach complies with the CORBA philosophy, by promoting modularity and reusability. Parallelization support may be provided by adapting an existing CORBA service, or by defining a new service for object groups, as we did in this thesis.

6.3.4 Evaluation of the Different Approaches

This section presents an informal comparison of the three different approaches based on [51]. It should not be considered as an exhaustive survey of the pros and cons of each approach.

The comparison focuses on several different aspects: transparency, ease of use, portability, CORBA compliance, performance, and simplicity. Table 6.1 summarizes how three approaches fit these criteria. In the table, + (plus) means good, - (minus) means limited, and blank means satisfactory.

Transparency

Transparency hides groups to the programmer, by giving the illusion that the invocations are issued to single objects.

All integration approaches provide full client transparency. The invoker does not need to know that the invokee is a group, although it could benefit from this knowledge.

The interception approach enforces transparency. In contrast with the other approaches, it does not allow to access group information or e.g. set a load balancing strategy.

Approaches	Integration	Interception	Service
Transparency	+	+	
Ease of Use	+	+	+
Portability	–		+
CORBA Compliance	–	+	+
Performance	+	+	+
Simplicity	–		+

***Table 6.1**: Comparison of the different approaches how to integrate parallelism into CORBA*

The service approach can be configured with or without transparency. Transparency is provided by overloading e.g. by the typed event service and is also provided by our CORBA Distributor Service.

Ease of Use

Ease of use is an important consideration since it shortens program development time and it can make the application more robust and reliable. It is the aim of this third part of the thesis to simplify the development of parallel implementations by offering the use of object oriented middleware to an application programmer but still rely on efficient communication.

Transparent group support is easier to use, since it requires no explicit construct on the client side. The client does not care whether its partner is a server object or a Distributor Service.

The event service provides a familiar Publish&Subscribe programming model. Its interfaces are already standardized and well-known, making them easy to use by many programmers. Our Distributor Service combines the advantages of explicit group management and transparent invocations in a CORBA service.

Portability

The portability measures how independent a software component is from a specific ORB or architecture.

It is clear that monolithic built-in functionality is clearly not portable to other architectures. The integration approach adopted by Padico uses non-standard language-specific constructs by an enhanced Interface Definition Language causing application code to be also not portable.

With the service approach, both the Distributor Service and the application code using the service are portable to any CORBA compliant architecture, assuming that the code does not depend on implementation-specific ORB feature.

CORBA Compliance

A client or server program is said to be CORBA compliant if it conforms to the CORBA specification. The integration approach is not fully compliant since it modifies and extends the CORBA specification. This has partly changed now as CORBA 3 defines Object Groups. Using and implementing just these groups would keep the ORB compliant but only to the CORBA 3 specification. The ORB core has to deal with references to replicated objects, and the semantics of CORBA references are modified.

Both the interception and service approach are compliant (assuming that they do not rely on implementation-specific ORB features). The interception approach is completely decoupled from the ORB and only relies on IIOP constructs. The service is independent of the ORB core since it is used only through IDL-defined interfaces, and does not make assumptions about the underlying ORB implementation.

Performance

Systems that provide high-level abstractions are generally less efficient than low-level systems. In the context of group communication, performance also depends directly on the protocols.

The integration approach is probably the most efficient, since communication can be optimized in the ORB itself. There is no indirection when invoking servers.

The efficiency of the interception approach depends both on the ORB (time required for IIOP communication) and the underlying group communication toolkit.

The service approach uses the communication primitives of the ORB on which it runs. Therefore, its efficiency depends directly on the underlying CORBA implementation. Two indirections are also required when a client invokes a group. As we are focusing on bandwidth limited application this is not a problem.

Simplicity

We consider an approach to be simple and lightweight if it is adequate for the problem and performs what it is meant to do, without overloading the system with unnecessary features.

The current implementations of the integration and interception approaches make use of external group communication toolkits like MPI. Most of these toolkits do not provide adequate support for object groups as they deal with process groups, and software layers must be added for interfacing them with the CORBA world.

In contrast, the service approach, as defined by our CORBA Distributor Service, is lightweight and adequate for the problem. It provides only the required primitives and is built as an independent, optional CORBA component.

6.4 Distributor Service

Because none of the existing CORBA services offer the right abstraction for parallel processing we have elected to specify a new service for parallelization and data distribution support in CORBA and named it *CORBA Distributor Service* in the hope that a similar specification can be established as a CORBA common object service (COS).

CORBA's open architecture allows us to easily define and implement new services. The process of specifying a new service consists in isolating the requirements, choosing the right abstractions, and specifying the interfaces for these abstractions. The OMG has published guidelines for designing object services and their interfaces [108].

In the following sections I will discuss some of the details that were taken into consideration while designing and implementing the distributor service. The complete IDL specification of the proposed CORBA Distributor Service can be studied in Appendix D.

6.4.1 Design Requirements

The distributor service as an abstraction provides a *Distributor Channel* as its main instrument to decouple the servers from the client. Suppliers can generate deliveries and deposit it on one side of the distributor channel. It is the task of the distributor service to pass the delivery through the channel to a possible consumer (see Figure 6.8).

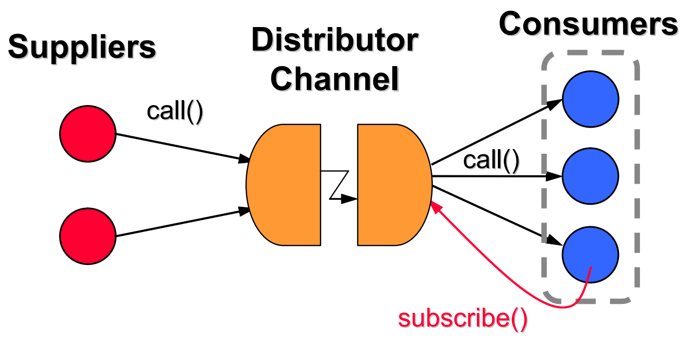


***Figure 6.8**: The Distributor Channel passes a delivery asynchronously from the supplier to one member of the consumer group.*

The following list specifies the goals that our CORBA Distributor Service shall achieve and its requirements:

1. **Asynchronous method calls**: The service asynchronously decouples the client from the servers (Publish&Subscribe) like an Event Service and delivers the requests as fast as possible to a possible consumer. The supplier of the service may not be blocked.

2. **Load balancing**: An inherent Load Balancer assigns deliveries directly to one of the consumers or partitions the parameter data collection and forwards them to multiple servers. The load balancing has information on the load of all consumers and can inform the supplier if the load is too high.

3. **Result forwarding**: The results are passed further to a result consumer object which can be the client again, but is not compelled to.

4. **Deliver at most once**: A delivery is passed with “best effort” semantic to exactly one consumer. Multiple deliveries must be avoided. When a delivery cannot be passed further the supplier has to be informed.

5. **Dynamic subscription**: Consumers as well as suppliers can subscribe and unsubscribe to/from a Distributor Channel during run-time without needing to inform its counterparts on the opposite side of the channel. This introduces a referential decoupling between supplier and consumer.

6. **Transparency**: The interface of the channel does not differ from the consumer interface. The signature of a method call to a consumer by the channel is exactly the same as the supplier call to the channel itself. The server interface is therefore transparent to the service interface as with parallel objects. The client uses the service transparently like it would use a single server object.

7. **Cascading**: Cascading of Distributor Services shall be allowed. This will mainly be given through the transparency.

8. **POA manager compliant**: The Service should be designed in a way that it can be hidden through a POA Manager. This would make the implementation even more transparent.

9. **Zero-copy aware**: Last but not least the Zero-Copy regime should be integrated. We intend to do that through object-by-reference calls with direct callback to the data objects to access data.

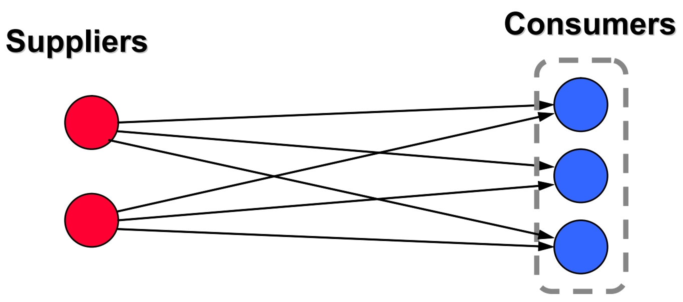
6.4.2 Prerequisites

The CORBA Distributor Service is primarily built on top of an existing COS Event Service [110] and not on a COS Notification Service. First the COS Notification Service is much wider specified and therefore much more complex in its implementation. As discussed in Section 6.2.2 it does not offer any advantages to the distributor service. Another reason that we chose the COS Event Service is that the used MICO distribution does not contain a COS Notification Service.

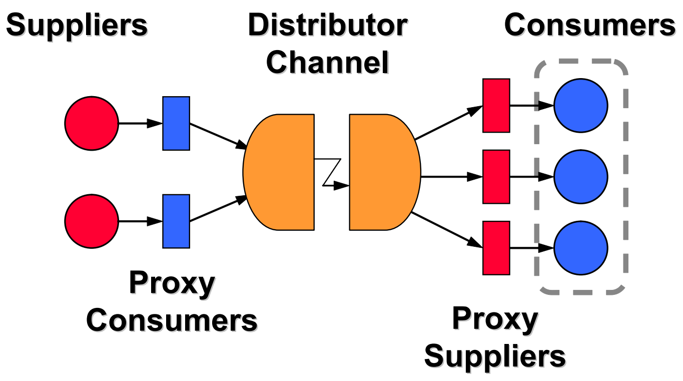
The kernel of the event channel is enhanced by a simple load balancing service as discussed in [76] and in [118]. Like this the service is able to consider the load of the single consumers while delivering a request. A further change of the "one-to-many" to a "one-to-one" delivery characteristic can also be incorporated by a small changes of the event channel implementation.

6.4.3 Event Communication

As discussed earlier in this chapter CORBA method calls are normally synchronous. A client calls a method of a known server and blocks until the server has processed the method and delivers the result. The generated CORBA request needs an unique and existing server object to call. This tight coupling between the client and the server is in many cases not wished or even not possible (see Figure 6.9(a)).



(a) CORBA method call



(b) Decoupled method calls

Figure 6.9*: 6.9(a) shows coupled Suppliers and Consumers with standard CORBA method calls, 6.9(b) shows decoupled Suppliers and Consumers using proxy objects that communicate through a channel.*

We therefore consider to use the channel technique as used in the COS Event Service for a complete decoupling between supplier and consumer. The communication is handled through a channel. Both the supplier and the consumer do not need to have references of each other they only need a reference to the channel. This is called referential decoupling.

The supplier then generates a request and passes this to the event channel. The respective method instantly returns without first waiting for the successful delivery of its request to the consumer through the channel. This concept is called asynchronity. It is now the channels task to distribute the request to one of the consumers.

Our channels do not support any quality of service facilities but as the notification service shows they can easily be extended.

The suppliers use the channel through a so called `ProxyConsumer`-interface, the consumer on the other hand uses the `ProxySupplier`-interface). This encapsulation of the channel easily allows to cascade several channels.

The event service specifies two communication models how events are delivered between supplier and consumer; a *Push model* and a *Pull model*. We only use the push model in the distributor service. There the supplier acts actively and pushes the requests onto the `ProxyPushConsumer`-interface of the channels. The consumer (i.e. the `ProxyPushConsumer`-interface of the channels) in passive until a request arrives which triggers a call to the `PushConsumer::push`-method of the consumer.

6.4.4 Integrated Load Balancing

Load balancing is the process of distributing load generated by clients to several available servers that are capable to process the requests. Load balancing is also used for application that need to support high reliability and zero downtime. This is achieved by replicating the servers and using them in parallel. If one of the servers fail the requests are just dispatched to the remaining servers until the broken server is online again.

The distributor service implements a load balancing service to dispatch the requests on a channel to the right server. The right means a server that implements the requested method and whose load is appropriately low for the task to be executed efficiently.

The complete IDL of the implemented CORBA Load Balancing Service is presented in Appendix C.

Balancing Granularity and Adaptivity

For a successful operation of a Load Balancing Service the granularity of the balancing is mandatory. There are two possibilities how such a service can operate. Either all the requests are handled individually or a client gets its definite server at its initialization and forwards all future requests directly to this dedicated server.

The balancing granularity of the Distributor Service is based on an individual request distribution. We prerequisite that the request contains its complete context information which is needed to deliver a request successfully. Because of this constraints the servers can act stateless.

The incidental computing overhead can be distributed adaptively or non-adaptively. The non-adaptive or fix load balancing strategies decide where to deliver a request without knowledge of the current process load on the compute servers and without feedback, e.g. *Round-Robin*- or *Random*-strategies. An adaptive method on the other hand considers at least an estimated load or even better decides on load information requested from and delivered by the server, e.g. *Least-Load*-strategy.

Adaptive algorithms are especially requested by inhomogeneous system environments where computers with different computing power or requests introducing different overhead are processed. However in many cases it is difficult to define an appropriate load metric. Another problem lies in the delay after which load information is available to the service. As the load on a server does not increase just by having been selected to handle a request but with the slightly delayed request processing start, this can introduce a scope for potential wrong decisions.

Design Overview

The load balancing service of the CORBA Distributor Service works with *Load Balancing Groups*. Different CORBA objects which typically implement the same interface are combined in such a group. The balancer then just chooses a member server of the appropriate group. The *Load Balancing Strategy* implements the method that is used to do the selection. Each member of the load balancing group is allocated to a non-ambiguous *Proxy Load Monitor*. This proxy enables the server object to deliver load information to the service (see Figure 6.10).

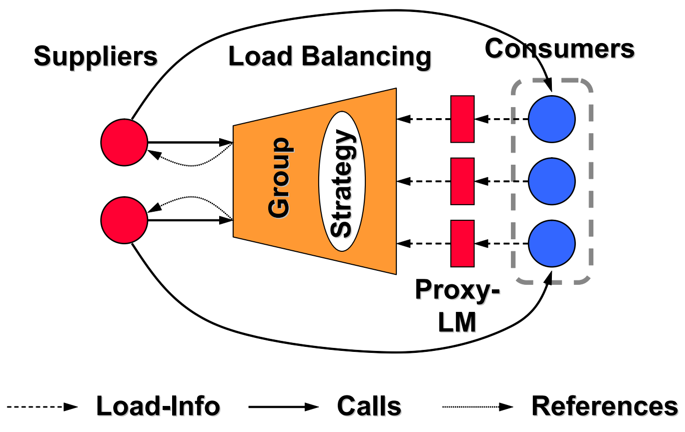


***Figure 6.10**: Overview of the Load Balancing Service*

Load Balancing Groups

The *Load Balancing Groups* provide the load balancing functionality towards its clients. A client contacts the group to get a server object reference. The load balancing group can provide additional information about the group like about special situations like e.g. not enough server capacity (`NoMembersReady`-exception) or missing server replicas (`NoMembers`-exception).

An object reference returned by the `Group::get_server`-methods does not guarantee the existence of this referenced object. This means that a client has to be aware of a `CORBA::ObjectNotExist`-exeption if that happens. An erroneous invalid object reference can e.g. occur on a server crash. On the other hand a given reference should always be used for a request processing to free temporarily locking (see Section 6.4.4).

The group itself does not control the state of a server directly. This monitoring task must be solved externally. The group just provides the functionality to list all the servers registered with `Group::list`.

This means that a client must pinpoint an invalid object reference and on occurrence desubscribe such a failed server reference from the group (`Group::remove_member`).

Proxy Load Monitor

Intelligent, adaptive load balancing needs to know about the processing load of its group members to dispatch a client request to the “right” server. This needed information can either be pulled from all the servers or be pushed by these servers.

The *Proxy Load Monitor* decouples the load sampling from the application specific task. By this existing servers can simply join a load balancing group. The server code does not have to be changed. The load sampling has to be done by a specialized component. In the case of more than one server on a single machine this load sampling can be done by a single load monitor.

The implemented functionality in the distributor service does not allow service-based load polling. Load information must explicitly be delivered by a server side daemon. This decision was motivated by the observation that only a server itself is always best informed about its load and the exact number of requests that it processes. A disadvantage of this method is as already discussed in Section 6.4.4 a slight delay between the load measurement and the arrival of the information at the load balancing group.

Another issues is the association of a load information message of a server to its associated group member. With the polling method this is implicitly clear but not with the push method. The group therefore delivers a `ProxyLoadMonitor`-object to the server on subscribing to the group with `add_member`. This proxy can uniquely be assigned to corresponding server. The proxy is deallocated together with the group member by the `Group::remove_member`-methode. This means that on a re-subscription to a group each time a new proxy load monitor will be allocated (see also Section 6.4.4).

Load Balancing Strategies

The *Load Balancing Strategy* is heavily dependent on the kind of application, on the method how the load information is made available and on the access pattern of the clients. The strategy is therefore not fixed into the load balancing group but can be changed even on runtime. Two built-in strategies are exported by the service. They can be configured via `Property`-objects:

Round-Robin Strategy Pure “Round-Robin” is the simplest an the best known practice. It always selects the member from a group that was least recently selected. This strategy accomplishes its requirement in many systems especially homogenous ones. The method is also simple because it does not need load information.

The built-in Round-Robin strategy is not parameterized and therefore does not support properties. It is selected by its name “RoundRobin” with the `Group::set _strategy`-method.

Least-Load Strategy “Least-Load”-strategies are the simplest kind of dynamic load balancing. It selects the member form a group with the least load.

This strategy has the late-update problematic discussed in Section 6.4.4. The timely delay between balancing decision and load report has a limited risk of wrong decisions but these risks can be mitigated with the aid of some additional rules. The following attributes serve for the solution of the late-update problematic:

- **“Default-load”-Property**: The “Default-Load”-property defines the load that may be allocated to a server at its initialization. The `value`-field must be initialized with a default value.
- **“Lock-until-info”-Property** With this property a server is blocked after it is selected until its load monitor delivers a load report. This circumvents multiple following requests.
- **“Per-balance-load”-Property** Instead of blocking a selected server object with the “Lock-until-info”-property the “Per-balance-load”-property allows to add a fixed additional amount to the current load. The `value`-field must contain this load value.

User Defined Strategies Besides the two built-in strategies the user of the load balancer can provide its own strategy. The strategy can be set via the group interface anytime. There are applications where it makes sense to select the strategy depended on certain environmental variables (e.g. time, overall-load etc.).

A user-defined strategy implements the `CustomStrategy`-interface and can be activated with the `Group::set_custom_strategy`-method. Some important points for the implementation of user-defined strategies are:

- Each strategy must manage the association between members and load information.
- It is essential that all strategies respect locking by `CustomStrategy::lock_member` (see Section 6.4.4).
- The return value of `CustomStrategy::next`-method delivers a reference of a registered, currently unlocked group member.

Member Locking

The implementation of the distributor service has shown that there is a need for temporary locks of members of a load balancing group. This is e.g. essential during an initialization phase of a round-robin load balancer where the channel is already used to dispatch requests while there first exists just one initial server. This server is inevitably always selected which leads to an accumulation of pending requests that can not be redispatched afterwards when more servers are available.

The `Group::lock_member` and `Group::unlock_member` functions allow to short-time lock a member and not use it for load balancing until unlocked. In contrast to the unsubscribing with `Group::remove_member` the proxy load monitor keeps its validity.

6.4.5 Distributor Channels

The central component of the CORBA Distributor Service is the *Distributor Channel*. Its job is to deliver load balanced requests by serving as a mediator between consumers and suppliers. It is also responsible for the referential decoupling (Section 6.4.3 and Figure 6.11).

The distributor channel hides an inherent instance of a load balancing group. On instantiation of a channel via the channel factory (`DistChannelFactory`-interface) the `create_distchannel_with_group`-method allows to explicitly assign a load balancing group (Section 6.4.4). Alternatively `create_distchannel` allocates its local load balancing group. But the assignment of an external group has two main advantages:

1. The locality of the group is not bound to the locality of the distributor channel.

2. As long as the interface `CosLoadBalancing::Group` is kept the implementation can be changed arbitrarily.

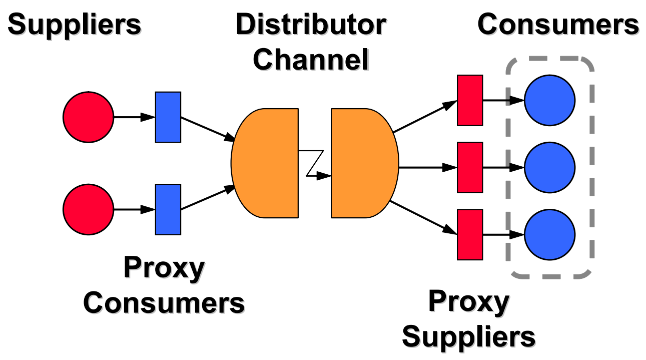


***Figure 6.11**: Components of the Distributor Service.*

The distributor channel does not provide any directly accessible functionalities but all the administrative interfaces are provided to configure and initialize the channels:

DistChannelAdmin::ChannelAdmin
The `DistChannelAdmin::ChannelAdmin`-interface is used to administer the distributor channel. In contrast to an event channel the distributor channel provides a load balancer for a "one-to-one" distribution characteristic. The strategy and further settings can be selected or adjusted by this interface.

DistChannelAdmin::ConsumerAdmin
This interface is merely used by the consumers. The method `obtain_push_supplier` creates a `ProxySupplier`-object (Section 6.4.6).

DistChannelAdmin::SupplierAdmin
Analogous to the `ConsumerAdmin`-interface suppliers can create `ProxyPushConsumer`-objects with the `obtain_push_consumer`-method implemented by this interface.

6.4.6 Proxy Objects

Neither suppliers nor consumers communicate directly with the distributor channel (the only exceptions are the three `Admin`-methods). Both supplier and consumer use an appropriate Proxy-Object of its counterpart that they initialize over the `Admin`-interface. The advantage of this indirection is that both parties can directly "see" the interfaces of its counterparts through this proxy. The channel inside the service is completely transparent (compare with Figure 6.9(b)).

The interface between proxy and channel is dependent on the implementation and not accessible to the programmer. It provides methods to register, connect and to lock proxies. The proxy consumer provides methods to accept deliveries.

Both the `ProxyPushConsumer`- and the `ProxyPushSupplier`-interface support a `disconnect_push`-method. The call of this method disconnects the link between caller and proxy. For a supplier this means that it is not further able to push deliveries until it is reconnected to the proxy again (`connect_push_supplier`) whereas for a consumer the disconnect also invalidates the proxy load monitor. Internally the proxy is removed from the load balancing group (Section 6.4.4). On a reconnect a new monitor is allocated.

6.4.7 Asynchronous, Decoupled Delivery Communication

The most important reason for the choice of the COS Event Service IDL as the origin for the CORBA Distributor Service is its decoupling characteristic. The referential decoupling is given by the proxies. The whole functionality of the call decoupling is therefore hidden in the implementation of the proxy supplier interface.

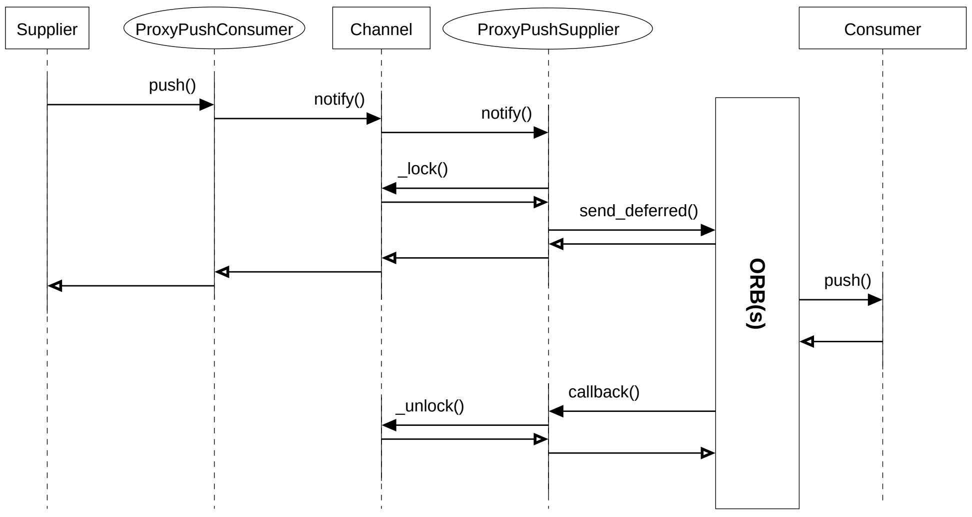


***Figure 6.12**: Call diagram showing the operating sequence of a consumer method call initiated by a supplier.*

Figure 6.12 shows the sequence of a method call as it is triggered by a supplier for the delivery of a message to a consumer. The first three method calls forward the delivery synchronously to the `ProxyPushSupplier`-object. This proxy object first temporarily signs off with the channel (locking, see Section 6.4.8). Afterwards the proxy generates a `CORBA::Request`-object which is sent via the “Dynamic Invocation Interface” (DII). For the sending the non-blocking `send_deferred`-methode is used in contrast to the normally used synchronous interface. CORBA does not specify secure asynchronous method invocations with the static invocation interface where only a oneway method could be used. But this method is specified as unreliable as it implements just a best effort functionality.

The instant return of the deferred call allows the instant termination of the chain of called methods back to the supplier while meanwhile the in the ORB deposited request is passed further to the consumer with the `PushConsumer::push`-method.

One of the parameters that are passed with the `send_deferred`-method is a reference to the proxy itself. As the `ProxyPullSupplier`-class is among others derived from `CORBA::RequestCallback` it can be installed as a callback object into the ORB-loop. This callback is issued by the ORB as far as the request is processed by the consumer or an error occurred. It deletes the lock and activates the consumer again.

6.4.8 Locking

The distributor channel automatically locks all the PushSupplier proxies which have issued an asynchronous request per default. This means that such consumers are not allowed for further deliveries (see Figure 6.12). This restrictive scheme is optional and can be turned of. But numerous experiments with distributor channels in cooperation with the transcoder application presented in Chapter 7 have shown that with very compute intensive jobs it makes sense to not issue more than one task at a time. The locking further allows some sort of synchronization of suppliers and consumers.

The locking mechanism of the distributor channel uses the locking functionality of the load balancing service to easily provide the functionality (see Section 6.4.4).

6.4.9 POA Manager

The Portable Object Adapter [116] defined by the OMG, which is compulsory since the CORBA 2.3 specification, was designed to meet several goals, mainly the portability of the object adapter between different ORB products. It provides:

Policies
: A POA based application can instantiate multiple distinct POAs. Policies allow configuring each of these POAs to match the application requirements.

Transparent Activation
: A POA can generate object references without really instantiate and activate the object. The activation is done on arrival of the first requests.

Multi-Object-ID Servant
: The POA allow a single servant to support multiple object identities simultaneously.

POA Manager
: Each POA object has an associated POAManager object which may be associated with one or more POA objects. A POA manager encapsulates

> the processing state of the POAs it is associated with. Using operations on the POA manager, an application can cause requests for those POAs to be queued or discarded, and can cause the POAs to be deactivated.

An interesting variant how the distributor service can be integrated into the ORB behavior would consist in providing a specialized portable object adapter that implements the distributor service functionality. We would then have the standard object adapter for standard objects and a dedicated " balancing and forwarding distributor adapter" for objects that are members of a load balancing group. This approach would be even more compliant with the CORBA way of programming and would encapsulate all the distributor facilities in a single object adapter component.

6.5 Experiences

With the implementation of a large part of the COS Load Balancing Service proposed by [76] a simple but flexible and portable service could be provided for MICO. The load balancing strategy which is responsible for dispatching the requests to the "right" server can be affected and even exchanged on the fly. Besides the two built-in characteristics, a simple round-robin and an adaptive least-load method, an own strategy can be implemented and be used like a plug-in.

None of the existing CORBA Services could match our requirements for a parallelizing service which are:

- asynchronous, decoupled calls,
- request load balancing with load monitors
- deliver at most once,
- dynamic publish & subscribe,
- transparency.

We therefore designed and implemented the CORBA Distributor Service that comprises all these desired attributes. In the evaluation this Distributor Service showed its potential impressively and performs very reliably.

The referential integrity of delivery suppliers and consumers even makes the service convenient for applications outside its dedicated goal platform, the clusters of PCs. We evaluated the architecture also for a small computational grid consisting of a large number of inhomogeneous computing resources spread all over our computer science department. The cascading facilities of the service made it possible to bind some weak machines together into a respectful grouped computing resource by just adding another service.

In the next chapter we look deeper into an application that uses the CORBA Distributor Service to parallelize its jobs and scale beyond computing resources available in a single server and still adheres to the zero-copy regime.

7 Zero-Copy Distributor Framework Evaluation

To evaluate the effectiveness of the Zero-Copy Strategies in our TCP/IP implementation and the CORBA middleware as well as the functionality of our Distributor Service we use a framework for the design of an application. Our Distributor Framework is used to implement an MPEG2-to-MPEG4 Transcoder that is cleanly modeled with distributed objects.

The chapter first outlines the architecture of the *Distributor Framework* which solves the following issues: (1) it first allows data to be read or deposited into the framework from several sources, then (2) a processing module, that is pluggable into the application, processes the data that lies in special data chunk buffers, and finally (3) the data can be exported to or fetched from different destinations.

The distributor framework is designed to work in conjunction with our *Distributor Service* presented in Chapter 6. The framework is used at three places. The first place is at the data source which is represented by a first process that imports the data, preprocesses it and makes it available to the processing modules. Then the data is sent to or fetched by a second module that processes the data according to its specification. This step which can be very compute intensive can be parallelized by the distributor service. The third process where the framework is used is a collector that gathers the results and further exports it to a consumer.

In a second part of this chapter the distributor framework is used to model a real-time MPEG2-to-MPEG4 Transcoder by just introducing specialized data Input-, Processing- and Output-modules. The video data streams that consists of a huge amount of images (or video frames) that are either directly grabbed form a HDTV frame grabber or read from a disk file or extracted from a DVD MPEG-2 stream is distributed by CORBA requests over a Gigabit Ethernet using the Zero-Copy TCP/IP sockets of Chapter 4 and the optimized Zero-Copy ORB introduced in Chapter 5. The distribution is implicitly handled by our Distributor Service.

7.1 Distributor Framework

7.1.1 Architecture

The design of a *Distributor Framework* around our Distributor Service was developed in several implementation and refactoring steps. The presented architecture took shape after we analyzed our MPEG Transcoder application as well as two similar application patterns that transmit huge amounts of data. The framework reduces all the logical conceptual components of a parallel data processing pipeline into one single tool.

The aim of the distributor framework (see Figure 7.1) is to provide a component that is capable of organizing three tasks in an application:

1. **Data Input**: The data import shall be allowed from many different sources, e.g. disks, pipes or CORBA-requests, and it shall even be allowed to deposit the data directly from outside.

2. **Data Processing**: The data processing component shall be pluggable and exchangeable. The component works on special buffers that allow an efficient communication between the components (with the Import- and Export-module).

3. **Data Export**: Similar to the input module the data export provides several possibilities. It also makes data available by CORBA objects references and allows to fetch it from external processes.

The architecture of the distributor framework is based on a central *scheduler-object* (Section 7.1.4). This scheduler object references an *input-*, a *process-* and an *export-module* (Section 7.1.3) and is responsible that all these three modules are called and active regularly.

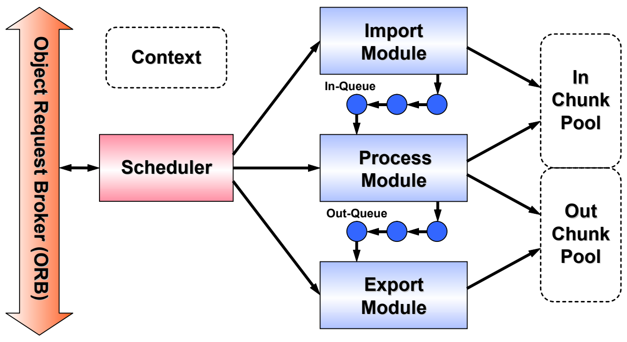


Figure 7.1: *Overview of the components of the Distributor Framework.*

To handle internal data communication between import- and process-module as well as between process- and export-module they share a pool of data-buffer objects, so called *chunks*. Chunks can be efficiently allocated and deallocated freely at discretion. A *pool* *manager* implements this efficient memory buffer management (Section 7.1.2). Process-modules that allow an “in-place” processing of the data (like e.g. the ProcessNull-module or the ProcessSort-module described in Section 7.3.3) the framework is optimized to allocate just one pool that will be shared to all the modules for efficiency reasons. With this an inherent data copy can be prevented and a zero-copy architecture on the application layer is possible.

The modules communicate the chunks via two separate FIFO-queues. Following our zero-copy regime these queues do not store the chunks themselfs but only references to data (IDs) that are queued and passed on. The queues serve as synchronization buffers between the particular modules providing enough buffering to decouple the processing steps.

To keep the interface of the pluggable modules simple, all the required parameters, references, counters etc. are put together into a form of *context-object* which is referenced by all the components. This also simplifies the configuration of the final application.

The only object of the framework implied and required by CORBA as encapsulation is a scheduler. And therefore the scheduler is the only component which can be accessed from outside via the ORB per default. All modules, queues and chunk-pools are handled like standard C++-objects by the scheduler.

7.1.2 Chunk Pool

During the initialization phase the `ChunkPool`-object allocates all memory resources needed. All these chunks in the pool are page aligned and each chunk is represented by a unique chunk descriptor which contains an associated chunk-ID, the exact size and a pointer to the buffer.

A chunk can be allocated with `createChunk(int chunkID)`. The `chunkID` must be unique. With this ID a pre-allocated chunk can then be taken from the pool by `getChunk` very efficiently. When a chunk is not needed anymore it must be released and returned to the pool with the `recycling`-method.

The chunks can be used to contain CORBA `sequences of octets`. But as CORBA provides inherent memory management for its standard data types this feature must be turned of to allow for our own management with zero-copy. If not turned off the CORBA ORB would deallocate the buffer once the object is not referenced anymore which would lead to unpredictable behavior of the chunk pool.

7.1.3 Module Interface

All the modules introduced in Section 7.1.1 are derived form the `Module`-interface. The scheduler uses this interface to interact with the module-objects and presumes an implementation of the the following methods:

int Module::open()

After all the three modules (input-, process- and export-module) are created this method is called. The module then initializes itself (memory allocation, connection establishment to other modules, open files, initialize libraries, subscribe to the ORB or a potential distributor service etc.). The ORB-loop is not yet running at this moment so requests cannot be triggered.

int Module::work_pending()

Before the scheduler calls the `run`-method of a module it tests whether the module is ready to perform tasks. Even if this method seems to be unimportant (`run` could just return) it has a critical effect to the behavior of the scheduler (see Section 7.1.4).

int Module::run()

The `run`-method starts the module. Dependent on the module type other conditions have to be checked.

int Module::close()

Shortly before termination of the transcoder the `close`-method is called. This instructs the module to free all its resources, to close its connections and files, to unsubscribe from channels and to terminate.

In Section 7.1.1 the scheduler was depicted as the only object which is connected with the ORB. This must be relativized at this point: Even if the scheduler is the only object whose referencing by the ORB is compulsory the module is not detained to connect its own objects with the ORB to react on external events or to provide CORBA servant facilities.

Import Module

The import-module is used to read the data into the chunks. It therefore allocates a new chunk from the pool and stores the data in it. The data can be read form disk, directly from a frame grabber, from a pipe or received by a zero-copy ORB request from another module. The chunk is then passed to the process-module by putting the chunk-ID into the inbound-queue of the framework. A call of the import-module (`run`-method) should at least allocated one chunk and put it into the inbound-queue.

There are different possibilities how the import-module accesses the data. The simplest is to just read the data from disk or a frame grabber. But the interesting mode is to get an CORBA object reference and to access the data in the depicted object directly via the zero-copy ORB. This makes the inter-framework communication over network very efficient, in many times even more efficient than accessing a local object that might be stored on disk.

To summarize the tasks of the import-modules:

1. allocate a new chunk from the pool for incoming data (in-pool)
2. read the data and store it into the chunk
3. append the chunk-ID to the inbound-queue of the process-module

Process Module

The process-module gets chunk-IDs from the inbound-queue and accesses via the chunk-pool the corresponding reference to the chunk descriptor that contains the data.

The process module can work with one single pool or two separate pools. This depends on the type of processing the module is intended for. For process-modules that just do “in-place” processing of the data (like e.g. the generic ProcessNull-module which is just used to pass the data from the input-module further to the export-module) the framework allows to allocate just one pool that is shared to all the modules for efficiency reasons. In a video encoder the data cannot be processed “in-place” and the output must be written into a new chunk of an out-pool. These chunks are forwarded to the export-module by just adding the chunk-ID to the outbound-queue. When using two chunk-pools the process-module is responsible for the deallocation of the in-chunk.

To summarize the tasks of the process-modules:

1. get the next chunk-ID from the inbound-queue
2. ask the in-pool for the appropriate chunk descriptor that matches the ID
3. (only in the case of disjoint pools) allocate a chunk from the outbound-pool
4. process the data, (when disjoint pools, the data must at least be copied into an out-chunk)
5. append the chunk-ID to the outbound-queue of the export-module
6. (only in the case of disjoint pools) deallocate the in-chunks

Export Module

Analogous to the process-module the export-module gets the chunk-IDs from the outbound-queue and looks up the chunk descriptor in the pool. Then it exports the data and releases the chunk.

Similar to the import-module the export-module provides different possibilities to make the data available. To export chunks efficiently the module wraps the chunk objects with a CORBA sequence and passes the object reference onto this object further to another import-module. The data in the chunk object can then directly be accessed by another process which might be another framework-instance.

To summarize the tasks of the export-modules:

1. get the next chunk-ID from the outbound-queue
2. ask the out-pool for the appropriate chunk descriptor that matches the ID
3. export the data
4. deallocate the out-chunks in the out-pool

7.1.4 Activation of Objects and Scheduling

The main problem with the chosen architecture is the correct scheduling of CORBA-objects and standard traditional C++-objects.

Basically all the objects must be able to act autonomously. This lead us to the idea to implement a central event loop, that periodically activates all the objects. A non-preemptive scheduler will do as the objects pass the control back to the scheduler loop on termination. But this idea unfortunately does not fit with the CORBA model. With CORBA-objects this is not possible because all CORBA based server objects need a running ORB mandatoryly. If the ORB-loop would be stopped the requests cannot be dispatched anymore to the registered objects. The ORB-loop is responsible to listen for requests and invoke the requested operations.

On the other hand if an ORB-loop is started this loop is blocking (`CORBA::ORB ::run`) the process and waiting for requests. There is no possibility to trigger an external scheduler regularly within the same process.

External Event-Loop Integration into the ORB Scheduler

To bind the ORB-components into the central scheduler loop of the distributor framework (or the other way round) an integration of the framework-loop and the ORB-loop had to be found. The same problem occurs when a CORBA system has to be integrated into a graphical user interface framework (GUI) like an X11-event-loop.

We looked at MICO [130] and ORBit [128] and both ORBs offer mechanisms to integrate external loops into the ORB scheduler. The MICO-ORB allows the registration of objects of type `CORBA::Dispatcher` into the dispatcher loop. All these dispatchers are called by the ORB-loop regularly to check pending events. On such events callbacks are initiated.

Another possibility would be to poll the ORB for pending events and to start the ORB with a non-blocking call that returns after the event processing. Here the question is raised how many requests the ORB-loop will process until returning. The specification says noting about what would happen if a an ORB gets requests continuously. It is not clear if in such a case the ORB-loop would block anyway as no idle states occur.

The Scheduler as a CORBA-Object

As a third possibility we considered and finally chose for our distributor framework is to encapsulate all framework-scheduling functions in a CORBA-object. This is depicted in Figure 7.1. With this architecture, the ORB-loop does not need to be intercepted and the modules can create CORBA-objects without problems. The framework-loop is contained in the scheduler object and can be triggered and activated anytime by a simple CORBA-request.

Admittedly the advantage of this concept raised another problem. While the framework-loop is running and processing, the ORB is blocking out requests, because MICO is designed as a single threaded ORB. This means that the scheduler has to interrupt itself periodically and pass the control back to the ORB-loop. We therefore provide a `work_pending`-method in all the modules to release control. As far as the scheduler detects all the modules to be in a "no-work-pending"-state, the control is released to the ORB.

This "no-work-pending"-state can only be modified by an external event. In this case, the scheduler is restarted by another CORBA-request. As the detection of such external events is not in all cases deterministic our framework-scheduler is started periodically by the portable object adapter using a special POA monitor. Another method would be to use a COS Time Service that triggers such events or just insert a backup-loop in a second process, that generates `Scheduler::run`-requests regularly. This solution with a second process also solves the initial activation of the scheduler elegantly.

7.2 Components for an MPEG Transcoder

The performance requirements of compression and decompression in video streams is highly asymmetric. While an MPEG stream can be decoded even with minimal computing power on a standard PC, the encoding is much more complex and highly compute intensive. MPEG encoding is so compute intensive that it is often used as a CPU benchmark nowadays.

This asymmetry of processing power implies that a real-time MPEG encoder must divide the job into chunks and distribute it onto a cluster of PCs to achieve the required frame rate. An commercial moving picture uses a framerate of 24 or 30 frames per seconds (fps).

The application we target with our zero-copy distributed processing platform is a real-time MPEG Transcoder. In a departmental project streaming over the given networking infrastructure has been studied at ETH during the 2002 Soccer World Championchip. The outcome was that it is indeed possible to multicast some streams all over a modern networking infrastructure and provide the community with moving pictures or fairly high resolution newscasts. The problem accounted in this project was that the used MPEG-2 encoder hardware does not compress the data far enough to multicast several streams at a time. MPEG-4 is the emerging new standard for Internet video and would solve this by allowing to send approximately 8 streams with the same bandwidth that was used for one single MPEG-2 stream. But when it comes to encode several streams with high resolution (HDTV) for real-time broadcasting there is no hardware around that can do this job. The experiment with the multicast of the soccer championship solved the problem with dedicated hardware and by cutting the size of the video screen and the framerate. As implementors of a scalable framework we offered the project to use our cluster for encode these streams in parallel and simultaneously test our distributor framework in a field test.

Figure 7.2 shows an overview of the parallel MPEG Transcoder. This section describes the components used.

7.2.1 Parallel Encoder Design

The transcoder is organized into three physically distributed components. These componens are depicted in Figure 7.2 and will be introduced and specified in the following sections. All the components are implemented in C++ and depend on MICO CORBA as their communication middleware.

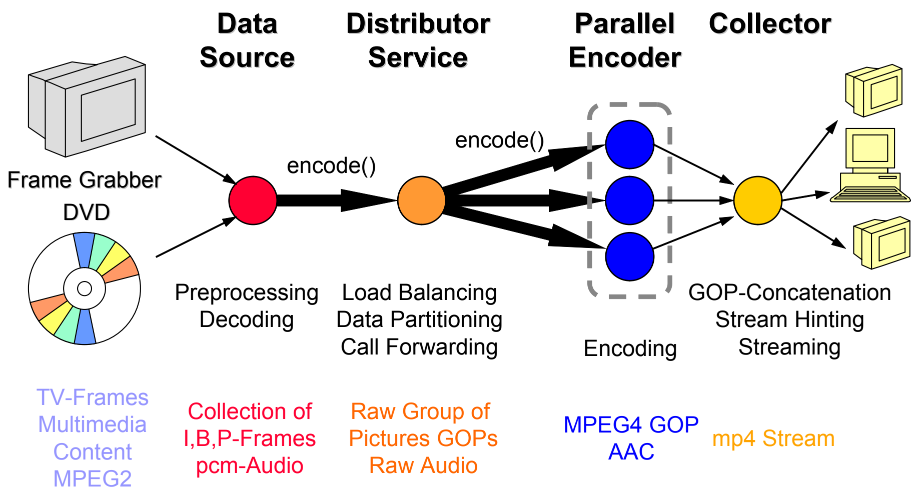


***Figure 7.2**: Overview of the MPEG Transcoder.*

The discussed transcoder application deals with video streams exclusively. The appropriate audio streams were handled by a separate converter running on a separate single PC since audio processing is much less expensive than video compression and in many cases the audio streams even could directly be copied from the MPEG-2 stream (see [69, 70]) while it is transcoded to the MPEG4 container format.

7.2.2 MPEG Transcoding

The transcoding from MPEG-2 video streams to MPEG-4 occurs in two steps:

1. The MPEG-2 stream is fully decoded into an uncompressed neutral format (YV12 in our case)
2. The uncompressed stream is reencoded and compressed into the MPEG-4 video format by a standard MPEG-4 encoder.

Some fundamentals of MPEG compression [89] are presented and summarized in Appendix A.

As mentioned, we use frames in YV12-format as the neutral video format. YV12 uses a different color space than RGB. If an image is stored in RGB format it must first be converted to YUV format.

The YUV format is downsampled slightly. All luminance information (Y) is retained; there are 8 bits per pixel of luminance information. However, chrominance information (U and V) is downsampled 2:1 in both the horizontal and vertical directions, which reduce the information by a 4:1 ration. Thus, there are 2 bits per pixel of U information and 2 bits per pixel of V information. This downsampling does not drastically affect quality because the eye is more sensitive to luminance than to chrominance information. Downsampling is a lossy step. The 24 bits per pixel of RGB information is reduced to 12 bits of YUV information, which automatically gives 2:1 compression. YUV frames are the prefered format for frame grabber input devices and can also be efficiently displayed in a viewer by most graphic card chipsets in hardware. It is also the input format of choice for many video encoders. (Section 7.2.7).

Like all video compression algorithms the advanced MPEG-4 compression scheme works with motion compensation and in most cases encodes just the differences between sequential frames. Therefore it can perform encoding efficiently only with sequences of frames. There are three types of frames: intra-frames (I), forward predicted frames (P), and bi-directional predicted frames (B). P- and B-frames are encoded relative to other frames. Each sequence starts with an I-frame which is more or less a compressed jpeg image. The following P-frames are encoded relative to the past reference frame which is an I- or P-frame, the B-frames are encoded relative to the past and the future reference frame. Motion vectors depict the translation of $16 * 16$ pixel macro blocks and an error term determines the difference. The search for the best motion vector (the one which gives the smallest error term possible and still leads to good compression) is the heart of any MPEG video encoder. The motion vector search is what makes encoders inherently slower than decoders. As less I-frames or as more P- and B-frames respectively as better the compression rate. But to minimize quality loss in the stream and to provide enough “random access points” that allow to synchronize a stream again when data gets lost on the way to the consumer, encoders normally place one I-frame each 20-30 frames. With 25 frames/s this leads to about one I-frame per second.

Format	Resolution [pixel@frames/s]	RGB Frame [KBytes]	YV12 Chunk [MByte]
QCIF	176 x 144 @ 15	76	0.57
CIF	352 x 288 @ 30	304	4.56
NTSC	720 x 480 @ 30	1037	15.55
HDTV	1920 x1080 @ 30	6221	93.31

Table 7.1*: Single RGB frame sizes and YV12-Chunk sizes needed to store one group of pictures (1 GOP/s) for different resolutions.*

7.2.3 Partitioning of a Video Stream for Encoding

At the beginning of the transcode pipeline the raw data must be imported. Different roots are available: YV12 coded data read form a hard disk or directly from a frame grabber or MPEG-2 video streams read from a DVD. The decoding of MPEG-2 video streams can be done by external tools like 'transcode' (Section 7.2.7). The transcoder can read input data both from files but also from named pipes.

The incoming YV12 stream first must be properly partitioned into chunks. The size of these chunks mainly depends on the size of the frames and the other attributes given by the MPEG encoder. The size of the video frames can be calculated by $frameSize = 3/2 * width * height$. The MPEG encoder needs a sequence of such frames to be able do encode a so called group of pictures (GOP). We normally work with one I-frame per 25 frames and 25 frames per second even if most US oriented standards use 30 (or 29.7) frames/s. This “downsampling” leads to $1/6th$ less data. For chunk sizes with different resolutions see Table 7.1.

7.2.4 Parallelization of MPEG-4 Encoding

The partitioned GOP data of a video stream can be passed further and distributed to the different MPEG-4 encoders. The distribution shall be transparent to the user and the partitioner and further the call of the distributor must not block the system. Such a blocking would lead to sequential execution and prevent all the targeted parallelism.

Another main issue in parallelizing is a fair and even distribution of the encoder tasks. To achieve a real-time encoding of a live stream the compute cycles provided by the servers should be optimally used. Even if we account for very efficient communication, chunks should only be communicated once. At a full HDTV resolution we obtain a chunk size of 93.31 MByte for one second of video and our 1 GHz Pentium III are able to communicate 110 MByte/s using a Zero-Copy ORB.

The functionality of the parallelization is accomplished by our Distributor Service (Section 6.4) which distributes incoming supplier deliveries to the registered consumers.

Format	Raw Data		MPEG4-compressed	
	Bit rate [MBits/s]	Storage [MByte/min]	Bit rate [MBits/s]	Storage [MByte/min]
QCIF	9.1	68	0.05-0.3	0.4-2.4
CIF	73.0	547	0.5-1	4-8
NTSC	248.8	1866	2-6	15-45
HDTV	1493.0	11197	15-30	113-225

***Table 7.2**: Bit rates and storage demands for different video formats raw and compressed.*

7.2.5 MPEG-4 Encoding

The YV12-frame sequences are encoded using the OpenDivX and XviD MPEG4 encoder library (see Section 7.2.7). Each sequence must be absolutely independent from other sequences. We therefore use a fix I-B-P-frame encoding schema. Whereas an adaptive scheme could improve the compression efficiency some more this has no influence on the quality. The incoming sequence does not contain any information regarding the order of groups of pictures (GOP). Table 7.2 shows some bit rates and file sizes that would be necessary for raw data as well as the bit rates and storage demand with the MPEG4-compressed streams for different resolutions.

The MPEG-4 encoding results in one second video streams that arrive out of chronological order since the encoding might take different amount of times depending of the frame complexity. Furthermore the encoding results are not of the same in size, therefore the resulting chunks contain different amount of data. These resulting chunks must be passed to a collector to sort the chunks and put them together into a consecutive stream.

7.2.6 Collector

The collector gathers the incoming chunk data. As the encoding time is dependent on the complexity of the frames each sequence is therefore given a unique ID by the partitioner. The collector can sort the chunks by ID and put them back into their original chronology.

An important problem which must be handled by the collector are lost chunks. It is always possible in distributed applications that a single component can fail. The collector therefore must recognize lost or delayed sequences reliably and take measures. For our demonstrator application we want to stream video data in real-time. Therefore handling lost data forces us to just means to drop such an errornous or missing chunk after a given delay. But once the collector detects losses it can also unsubscribe a failing encoder from our Distributor Service to prevent further sequences from failing or reinitialize the partitioner to resubmit a lost chunk.

After the sorting the collector can export a contiguous MPEG-4 stream. But this stream is a raw MPEG-4 video stream and it must be enclosed by a container format like VOB, AVI, Quicktime or MP4 to be able to be played by a viewer or be streamed over the network. The collector component must therefore encapsulate the MPEG-4 video stream (and optionally an appropriate AAC-audio stream) into the MP4 container format that includes synchronization as well as streaming hints. This format can then readily be multicasted accross a network and played by many clients.

7.2.7 External Components

Several Open Source codes of tools and libraries were studied and used for the implementation of the transcoder. Besides Linux and MICO the following standard software components were used.

transcode

transcode [117] is a text console video-stream processing tool. It supports elementary video and audio frame transformations. Some example modules are included to enable import of MPEG-1/2, Digital Video, and other formats. It also includes export modules for writing to AVI files with DivX, OpenDivX, XviD, Digital Video or other codecs. A set of tools is available to extract and decode the sources into raw video/audio streams for import and to enable post-processing of AVI files.

transcode can be used as an application in a so called UNIX pipe without problems. The command itself provides many command line options that can cover a broad spectrum of video streaming operations.

The transcode framework itself is similar to our framework but uses another container. The whole functionality is linked dynamically from shared-libraries on startup. Each transcode instance consists of an import- and an export-module which can be combined arbitrarily. But communicating over UNIX pipes leads to local copies and large overheads and the framework is restricted to one host. However the modular architecture of transcode influenced the design of our Distributor Framework largely.

MPEG4IP

MPEG4IP [98] provides an end-to-end system to explore MPEG-4 multimedia. The package includes many existing open source packages and the ”glue”to integrate them together. It is a tool for streaming video and audio that is standards-oriented and free from proprietary protocols and extensions. The MPEG4IP source code is provided under the Mozilla Public License.

Provided are an MPEG-4 AAC audio encoder, an MP3 encoder, two MPEG-4 video encoders, an MP4 file creator and hinter, an IETF standards-based streaming server, and an MPEG-4 player that can both stream and playback from local file. For the collector of our transcoder uses the mp4 creator, hinter and the streaming server provided with MPEG4IP.

The development is focused on the Linux platform, and has been ported to Windows, Solaris, FreeBSD, BSD/OS and Mac OS X.

OpenDivX and XviD Codec

XviD is the latest Open Source MPEG-4 codec that is compliant to the ISO standard. XviD is the first MPEG-4 encoder being developed under the open source model, much like the LAME MP3 encoder, and gives surprisingly good results despite the early alpha status. The compression rates achieved by the encoder are 8-10 times better than that of a current MPEG-2 codecs.

XviD is based upon the efforts made in the Open Source project ”OpenDivX” initially sponsored by DivX Networks Inc. Unfortunately they shut down the project and closed the source with all the ideas and work spent by many coders from the net. But some of the original contributors wanted to continue the work, collected the latest sources they had and brought them together to continue with that work. Since the start-over, XviD is developing very fast and is evolving into a very high quality codec.

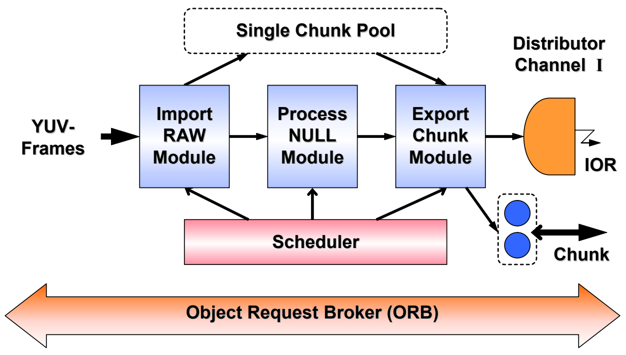
7.3 Implementation of a Distributed MPEG Transcoder

This section discusses the implementation of our MPEG Transcoder regarding to the specification in Section 7.2. It uses our distributor framework introduced in Section 7.1 which simplifies the implementation of the application significantly. The framework works with import-, process- and export-modules that have to be implemented according to the job they have to do. Using the framework therefore reduces the task to the development of these modules. The framework itself relies on our new CORBA Distributor Service discussed in Section 6.4 to parallelize the jobs to a cluster of PCs. Figure 7.3 shows the configuration we aim at.

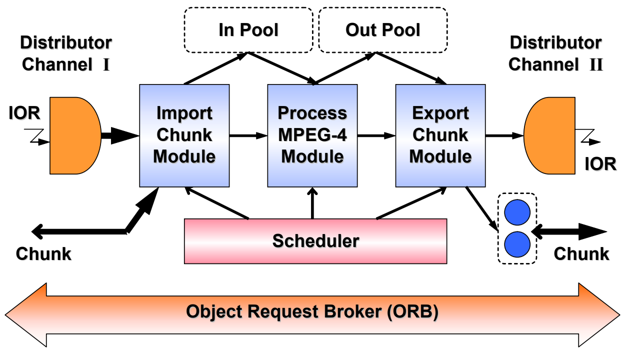
7.3.1 Chunk Communication

Reading and writing data chunks to disk is a relatively simple and cheap form of communication with the outside. This is much different with the import and export of video chunks as they have to be encapsulated by CORBA objects, exported by the supplier and gathered by a remote consumer.

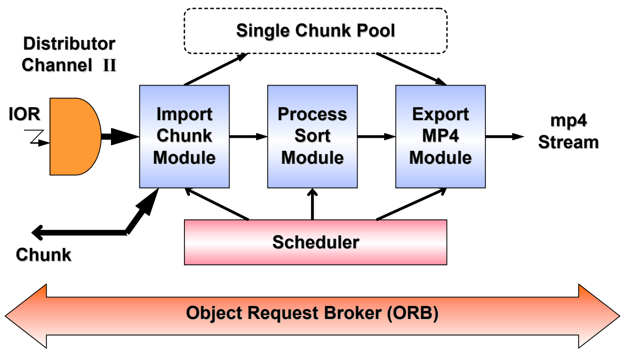
As already described in Section 7.1.3 and stated in Section 7.2.4, the distribution of raw video data that is needed for the encoding parallelization must be as efficient as possible. It must be implemented without in-memory copies and optimized zero-copy



(a) Data import, preprocessing and partitioning



(b) (Parallel) MPEG-4 encoding



(c) Sorting and MP4 stream export

***Figure 7.3**: The MPEG Transcoder requires the implementation of four special purpose modules: ImportRAW-, ProcessMPEG4-, ProcessSort- and ExportMP4-module. Additionally three standard modules provided by the framework are used: ImportChunk-, ExportChunk- and ProcessNull-module*

communication for data chunks must be used. As we know the requests are not sent directly to the consumer but they are sent to a mediator which forwards the objects to its definite destination. For large parameters this might be inefficient, that is why we implemented a callback strategy. The export-module encapsulates the data chunks that have to be exported by a CORBA object (`VideoChunk`-object) using zero-copy sequences (introduced in Section 5.6.1) and just passes the appropriate object references further. The CORBA objects are registered with the ORB so they can be accessed by their object references from outside. Figure 7.4 depicts this strategy.

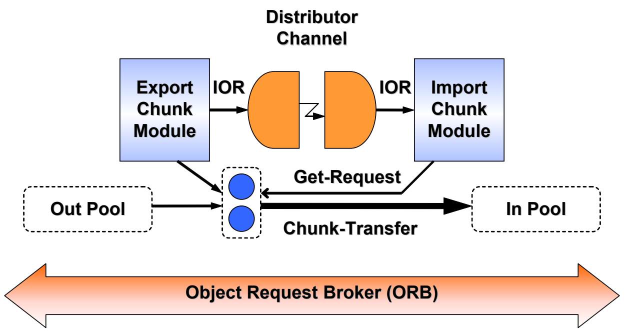


***Figure 7.4**: Chunk-transfer: The ExportChunk-module exports and distributes the IOR of the chunks that are encapsulated by CORBA-objects via the Distributor Service. The ImportChunk-module receives this IOR, requests the data directly from the ExportChunk-module and stores it into the prepared chunk allocated from the in-pool.*

This scheme allows the import-module as a consumer of our CORBA Distributor Service to access the data chunks (`VideoChunk`-objects) directly inside the export-module of the last framework instance and enables to copy the data form the remote chunk directly into the local chunk without any in-memory copies and without the detour over the distributor service.

7.3.2 Import Modules

The architecture as depicted in Figure 7.3 requires the implementation of two import-modules. The first module imports YV12 video data and is able to partition the data into single frames or groups of pictures (GOP) (Section 7.2.3). The second module imports chunks by accessing CORBA chunk objects as described in the last section. Each chunk (or partition) gets an increasing unique ID that is needed for sorting the partitions again after the processing. There is a third module that can be used to read the data directly from a frame grabber to allow real-time MPEG-4 streaming.

ImportRAW Module

This module reads YV12-data from disk or from standard input (`cin`) and partitions it into data chunks that are compatible with our zero-copy distributor framework.

ImportGrabber Module

Instead of reading the data from disk, this module makes a connection to a standard video interface and accesses the YV12 data directly on the frame grabber interface card. It also partitions the stream and stores it in the chunks provided by the framework.

ImportChunk Module

This module implements the `DistComm::Consumer`-interface of our proposed CORBA Distributior Service (for interface definition see Appendix D). Each delivery contains the IOR of a `VideoChunk`-object. This chunk is subsequently registered with the ORB and can therefore be accessed via CORBA by the ExportChunk-module.

As the receiving of a delivery is an externally triggered event the module calls the scheduler after a successful import.

7.3.3 Process Modules

The distributed MPEG Transcoder needs four process-modules that implement the principal functionality of the video encoder. These modules get the data chunks, apply their modification to the data and pass the result further to the export-module.

ProcessNull Module

The ProcessNull-module is the most simple of all process-modules. It just copies the data from the inbound-chunk to the outbound-chunk. This is very inefficient since this would introduce a local in-memory copy for no good reason. Therefore the module offers to use just one single chunk-pool. With that optimization the data is not touched anymore and only the chunk-ID is passed from the inbound-queue to the outbound-queue.

ProcessMPEG4 Module

The ProcessMPEG4-module transforms the groups of YV12-frames in the chunk into an MPEG-4 video stream by using the XviD-library [164] as a successor of the successful DivX-library (Section 7.2.7). The resulting streams are not necessarily of the same size so the data chunks might not be fully used.

This is the module that needs the most computing power and is therefore a candidate for replication in the parallel version.

ProcessSort Module

We have already discussed that the sequences encoded at different encoders might be returned out of order. The collector's task is to sort the chunks again which is exactly the job of this modules. It is the only module that operates on the whole inbound-queue, all the other modules use the queue as a FIFO queue and access only the first element.

The "lost chunk"-problem introduced in Section 7.2.6 is handled by the following strategy: As long as the inbound-pool still contains free chunks, the collector waits as the lost chunk could still appear. But if the inbound-pool is filled up some missed chunks have to be dropped until the next chunk in row is found in the pool. Should the missing chunks later reappear they are dropped anyway as the stream generation has already proceeded further.

ProcessDVDDecode Module

The specification of our application includes the idea of MPEG2-to-MPEG4 transcoding as required in the application of streaming DVD content over the net. This module allows this by decoding the video object container (VOB) files of a DVD. The VOB file contains the MPEG-2 video streams and several audio streams in different languages and formats (mostly the compressed Dolby Digital or the lossless dts Digital Theater System surround formats) as well as textual info for subtitles. The ProcessDVDDecode-module separates the video stream and decodes the compressed MPEG-2 format into single YV12 frames using libraries of the transcode framework [117].

7.3.4 Export Modules

Besides a module that exports the video chunks to our CORBA Distributor Service the collector needs a module that concatenates the partial stream fragments. As the MPEG-4 stream fragments cannot just be concatenated without readjusting some information like e.g. timing tracks a separate module is implemented. The resulting stream is just an MPEG-4 video stream and cannot be played yet unless it is enclosed in a multimedia container format like e.g. MP4. MP4 is a candidate for a new standard format for MPEG-4 video. It is derived from Apples Quicktime format and is standardized in the MPEG-4 specification [70].

ExportRAW Module

This module reads the data chunks in the order given by the outbound-queue, concatenates them and writes the data into a file or a UNIX pipe (`cout`).

ExportMultiFile Module

Instead of concatenating the chunk data this module creates a new file for each chunk. The file names are derived from the given name plus the chunk-ID (e.g. “output-0197”).

ExportChunk Module

Upon initialization the ExportChunk-module creates another CORBA portable object adapter (POA). The POA is configured with the `USE_DEFAULT_SERVANT`- and `NON_RETAIN`-policy. All the chunks that have to be exported can therefore be represented by a single `VideoChunk`-default-servant [72, 130].

For the generation of IORs we waive the inherent POA functionality and derive the object-ID from the appropriate chunk-ID. The `VideoChunk`-default-servant can decide by means of this IOR which of the chunks is referenced and delivers the data back.

ExportMP4 Module

This module finally implements the collector functionality as described in Section 7.2.6. It mainly concatenates the MPEG-4 stream fragments into a contiguous stream and embeds the stream into the MP4 container by using the MPEG4IP library [98] (Section 7.2.7) . The stream hinting allows to directly broadcast the resulting mp4-file as a stream by a video streaming server like e.g. Darwin [74].

7.3.5 trans command

The **trans** component is the unix wrapper around our parallel encoding framework and can be started by using the `trans` command line tool. The many setting allow to configure the framework, especially the modules that shall be used as described in the last section. For a detailed description of the syntax and the options see the “manual page” in Appendix B.

7.3.6 Example

These examples show how the transcoder is started up and put to work in the configuration given by Figure 7.3.

As a prerequisite for using the framework in a distributed environemnt we need a running COS Naming Service as well as our CORBA Distributor Service:

```
> ./imr create NameService poa ./nsd
    IDL:omg.org/CosNaming/NamingContext:1.0#NameService

> ./distd &
```

First we start an import- and partitioning instance. In this example the data import from the frame grabber device as well as the partitioning is provided by the ImportGrabber-module (`-ig`) and we do neither need a process-module nor a second chunk pool. So we configure the ProcessNull-module (`-pn`) and optimize the application to use just one single chunk pool (`--pools 1`) to prevent copying. The export shall make the chunks accessible for an encoder object that is why we configure the ExportChunk-module (`-ec`) and give the name of the distributor channel we want to use (`--outchannel toEncoderChannel`). The YV12 stream does not contain any hints about the frame rate nor the frame size this has to be given as an argument. We use an NTSC resolution of 720*480 pixels (`-w720 -h480`) and a frame rate of 25 frames/s (`-r25`). The partitioner shall group 25 frames into one group of pictures (GOP) (`-f25`).

```
> ./trans \
     -ig -pn -ec --pools 1 \
     -w720 -h480 -r 25 -f 25 \
     --outchannel toEncoderChannel
     - -
```

For demonstration purposes we show here the start of just one single encoder instance. But as the major goal is distributed execution in parallel the framework allows to start many instances of the encoder on different hosts. The import is done by the ImportChunk-module (`-ic`), the processing uses the ProcessMPEG4-module (`-pe`) and the export uses the same module as the first framework instance, the ExportChunk-module that export to a second channel of the distributor service (`-ec`). This time we have to specify the name of both, the input- (`--inchannel toEncodeChannel`) as well as the output-channel (`--outchannel toSortChan- nel`). And again the video settings must be given for the encoder. Even if they can be accessed through the chunk-object and are therefore even communicated between the frameworks but they are already needed at the initialization state of the chunk-pools.

```
> ./trans \
     -ic -pe -ec \
     -w720 -h480 -r 25 -f 25 \
     --inchannel toEncodeChannel \
     --outchannel toSortChannel
     - -
```

The collector instance finally imports the chunks again via the distributor service with the ImportChunk-module (`-ic`), but this time form the 'toSortChannel' (`--inchannel toSortChannel`). The sorting task is done by the ProcessSort-module (`-ps`). Again the ProcessSort-module can waive on a second chunk pool (`--pools 1`). The last module, the ExportMP4-module (`-es`), then does the rest. It concatenates the MPEG-4 video streams appropriately and embeds it into an MP4 container that contains further hints for streaming. The output is store into a file (`/tmp/starwars.mp4`).

```
[] ./trans \
   -ic -ps -es \
   -w720 -h480 -r 25 -f 25
   - /tmp/starwars.mp4
```

7.4 Evaluation of the Results

The design and implementation of a distributed MPEG Transcoder lead us to conceive a framework for distributed and parallel data processing with CORBA. The result is a modular, extensible, intuitive and simple CORBA Distributor Service and a framework to use the service. The software components are general enough to implement any distributed parallel application that deals with large amounts of data in a CORBA environment.

The distribution and parallelization of the processing is achieved by using our CORBA Distributor Service. This service schedules calls to different machines evenly by an internal load balancer and decouples the data supplier from the consumer.

The issue that lead to this application and framework development was the demonstration for optimal software efficiency using zero-copy implementations for all the middleware and underlying networking software. The framework deals with large amounts of data that is distributed to different processing nodes over the network and zero-copy data management is essential to achieve reasonable performance. Zero-copy can be achieved by using a highly optimized CORBA ORB and wrapping data into CORBA sequences of octets that can be communicated very efficiently. Our modified ORB relies on a zero-copy TCP/IP stack that offers the standard socket API as its interface. Both these prerequisites are introduced and discussed by this thesis.

Another issue that must be considered when optimizing software efficiency is raised by the architecture of the distributor framework. Although the service is transparent to the application implementor one must keep in mind that the calls including their parameters are sent to a service which forward them to a target server that does the processing. For large amounts of data such a scenario would introduce copies and way too much overhead. The distributor framework therefore exports the wrapped data as CORBA objects and passes just a CORBA object reference as a parameter to the call. This is similar to a local call-by-reference passing just a pointer to a structure instead of copying the structure itself. CORBA assures a global name space for those references and CORBA object references act as the data pointers. They permit that the data can be accessed from the outside directly from any processing module that needs to access the data. This is a form of callback function that is hidden by the CORBA middleware.

As a technology demonstrator the MPEG Transcoder `trans` is written for this distributor framework and can therefore be implemented by just adding special purpose modules to the framework base application. The overhead and complexity introduced by the parallelization and data distribution is completely hidden to the implementor. And so is the overhead generally introduced by network communication via middleware. We already showed a performance achievement of a factor of 10 for an optimized ORB communicating through a zero-copy operating system stack versus the original ORB. Therefore the entire performance gain is posed to our application.

The resulting tool is a command line based application for transcoding MPEG-2 DVD movies to the new MPEG-4 standard offering much better compression rates. The compute intensive encoding operation can be sped up to real-time operation by just increasing the compute power by adding some more PCs to a cluster and starting another encoder task on these machines. The initialization of the entire software system, registering to our new CORBA Distributor Service and the load balancing of jobs is done automatically at run-time. 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.

8 Conclusions

This dissertation dealt with novel software architectures providing efficient communication facilities over commodity Gigabit/s networks, in particular over Gigabit Ethernet. To ease the development process of parallel applications on Clusters of PCs a modern component and object oriented distributed object middleware platform is highly desirable. The thesis deals also with efficient software support for such a framework preserving the full communication speed of the lower software layers.

The principal intellectual contribution of this dissertation is its investigation of communication software efficiency. We define this efficiency as the ratio between the bandwidth that a hardware is able to deliver versus the bandwidth that is provided to the user application by the software system.

This dissertation first shows that respecting the best possible granularity, the communication in high speed computing platforms can be very efficient especially when using specialized hardware and optimized communication system software. We identify preconditions and techniques to extend this level of efficiency to standard software and commodity hardware in clusters. We further identify the main performance problem with Gigabit Ethernet, namely that transport protocols are needed whose operating system implementations introduce copies in the local memory system. As todays network bandwidths match the memory copy transfer performance, this memory copy turns out to be the most expensive bottleneck in high bandwidth communication with Ethernet.

The dissertation therefore argues that many applications that are easily parallelizable for clusters are limited by the communication bandwidth of the commodity networks and it claims that *Zero-Copy* is the most essential principle to fill the gap between hardware bandwidth and what is delivered by current software systems.

Since network bandwidths and memory copy transfer speeds are of the same magnitude in modern systems it is absolutely necessary to eliminate all in-memory copying and to design the software for zero-copy communication to obtain the best possible performance. All data that arrives in memory must be put to the right position when it is pulled from the network device, even before it is requested by a receiver.

Todays complex systems need to be constructed from many standard software components involving different layers of middleware that ease the development and cut the applications complexity into manageable parts. As a consequence we need a systematic software implementation strategy that applies the zero-copy principle to all the involved layers.

In our understanding a software implementation with zero-copy really means *zero* data copies through *all the involved layers* between a user application and the hardware. Partial so called ”zero-copy” solutions at the OS-layer do not qualify. This means that no copy may occur in the communication system including the hardware drivers and the network and transport protocols as well as by crossing the kernel-user boundary involving the socket interface. Middleware that provides an easier and more homogeneous system abstraction and some useful functionality to the application may not introduce any copies as well.

Wherever a copy may occur, one copy is always one copy too much and looking at the performance it will break the whole zero-copy principle. While additional copies can be cheap the first copy is always the most costly one, considering that data might be readily available in the cache after a first access in a system.

As the system software layer and the middleware layer have to be discussed separately and can not be merged into one, the contribution of the thesis are represented in two main sections: zero-copy with commodity communication hardware and zero-copy with distributed object middleware. The contributions for both parts are summarized once more in Sections 8.1 and Section 8.2.

As it is not easily possible to extend the simple message passing system to a development environment for distributed object oriented programming without modification to the message passing interface standard (MPI) we rather chose to start with the rich functionality of CORBA and optimize it for better performance. We transparently utilize the ORB functionality in a high performance system so that high bandwidth communication can be handled efficiently. Distributed object middleware like CORBA as an alternative to the MPI makes additional flexibility and the possibility for object oriented component design methods as well as the usage of design patterns and frameworks available to the developers.

In a third part the dissertation proposes ideas on how to use distributed objects for high performance parallel processing. We observed that none of the currently available CORBA components is able to completely satisfy the requirements given by the described MPEG-Transcoder application and therefore we had to introduce the design of a new service which we named *Distributor Service*. To allow fast development we further present a *Distributor Framework* that is used to implement an MPEG2-to-MPEG4 Transcoder. This contribution is discussed in Section 8.3.

8.1 Efficient Defragmentation with Ethernet

The small maximal packet size of standard Gigabit Ethernet prevents previously known zero-copy protocol optimizations unless IP packets can be fragmented most efficiently at the driver level without involving any additional data copies. Accurate fragmentation and defragmentation of packets in hardware remains impossible with most existing commodity Gigabit Ethernet network interface chips unless a *speculative approach* is taken.

With speculation techniques it becomes possible to implement true end-to-end zero-copy TCP/IP based on some simple existing network adapters. A substantial fraction of the peak bandwidth of a Gigabit Ethernet can be achieved for large transfers while preserving the standardized socket API and the TCP/IP functionality. The study of the speculation hit rates within two applications (TreadMarks SOR and a Oracle TPC-D benchmark) shows that misses are quite rare and that they can be further reduced in the highly regular Ethernets connecting the compute nodes of a Cluster of PCs.

Our speculative packet defragmenter for Gigabit Ethernet successfully relies on an optimistic assumption about the precise packet format, the integrity and the correct order of incoming packets using a conventional IP stack as fallback solution to deal with speculation failure. The driver works with the simple DMA (direct memory access) hardware of the network interface card to separate headers from data and to store the payload directly into a memory page that can be re-mapped to the address space of the communicating application program. All checks whether the incoming packets are handled correctly according to the protocol are deferred until a burst of several packets arrived. If the speculation missed the correct order or layout, some cleanup code passes the received frames to a conventional protocol stack for regular IP-processing.

The idea of speculation immediately raises the issues of improved hardware support for better prediction or for better speculation about how to handle incoming packets most efficiently. Two promising approaches are outlined, the first approach suggests a hardware extension of the network interface using a few simple protocol match CAM (content addressable memory) registers that classify incoming packets into two different categories: high speed zero-copy traffic and regular TCP/IP traffic. The match cam registers would control the selection of DMA descriptors used to store the packets in memory from different (at least two) descriptor lists. The second approach uses a dedicated protocol for admission control that guarantees exclusive access for one burst data stream.

Our implementation of a speculative Gigabit Ethernet driver with a speculative defragmenter is embedded in an OS setting with well known zero-copy techniques like “fast buffers” or “page remapping”. Together with those mechanisms a true zero-copy implementation of TCP/IP for Gigabit Ethernet has been achieved and measured. The implementation delivers 75 MByte/s transfers on 400 MHz Pentium PCs together with “fast buffers” support — a substantial improvement over the 42 MByte/s seen in the standard TCP/IP stack in Linux 2.2.

The benefits of a zero-copy communication architecture are not limited to high peak bandwidth numbers, but also include a much lower CPU utilization that will improve the performance of many application. On more current machines running at 1 GHz and providing an enhanced 64 Bit/66 MHz PCI I/O-bus the bandwidth achieved is 115 MByte/s while the CPU utilization stays at 30% compared to 100% with an unoptimized stack. Our Dolly tool for data streaming and disk cloning over Gigabit Ethernet was measured to perform 39% better with our zero-copy communication architecture than with a standard protocol stack.

A closer look at the success rates of speculation in the communication patterns of different applications indicates that either some additional hardware support or an admission control mechanism should be used to handle the bulk transfers occurring in larger applications. Still our approach of speculative processing in hardware linked to a fallback with cleanup in software seems to enable a set of new simple solutions to overcome some old performance bottlenecks in network interfaces for high performance distributed computing.

8.2 Throughput Efficient Middleware

Although heterogeneity in distributed systems is natural, most high performance distributed systems are characterized by some very limited heterogeneity. The components that are diverse can use a CORBA ORB, whereas components that are built using the same programming language and operating systems can use some simple software that applies a ”flyover”and bypass most of the standard CORBA operations and hereby save a lot of processing in terms of middleware overheads.

To achieve a zero-copy bypass for untyped data handled in CORBA applications we propose to use some standard CORBA method calls and introduce a *Separation of Control and Data Transfers* within the ORB. The separation of synchronization and data transfer is the key insight that permits better communication performance in parallel computers. The biggest speed improvements are caused by delegating buffer management to the application or if a middleware establishes the communication to this software layer in between. Looking at the structure of CORBA applications this means that the buffers are allocated and managed by the application or the stub and skeleton code generated through the toolkit of the according ORB.

As usual for CORBA applications the buffer management is handled by the middleware and it becomes possible to optimize this buffer management in the middleware implementation. This does not effect the user application while allowing for much faster communication. Instead of using parallelizing compilers which have knowledge about the communication pattern of a user application we rely on the knowledge that the programmer has an idea of the communication pattern generated in the ORB. Since we do not want to change the application interface and the synchronized client server messaging model of CORBA we introduce a decoupling of synchronization and data transfer inside the ORB, in th form of an IIOP communication with its partner.

The achieved performance results of this optimized ORB look very promising and the goal to optimize a CORBA ORB for direct deposit zero-copy operation was fully achieved. The throughput results on our optimized MICO ORB matches very well the raw TCP/IP-socket performance. That proves that the optimized ORB handles the new `ZC_Octet` sequences correctly by just passing them through the ORB while not introducing much overhead.

It is astonishing that the optimized ORB still comes close to the maximal performance of the much improved protocol stack using the zero-copy TCP. This demonstrates that our argumentation that zero-copy is the most essential technique to optimize for efficiency is true. The results also show the significant contribution of the idea to separate control and data transfers in the ORB.

As a contribution at a higher, conceptual level we prove that it is indeed possible to use CORBA middleware on top of a standard socket application programming interface and still adhere to the zero-copy regime. This was achieved and successfully demonstrated in a benchmark and an application. For large blocks our ORB achieves 550 MBit/s throughput on 400 MHz Pentium II PCs while the application still fully complies tho the CORBA standard. That means that the programmer work in the CORBA framework as he just defines the interface and generates the stubs and skeletons which map the server implementation with an IDL compiler. Given an optimal zero-copy system he is then ready to run his application efficiently on a distributed environment. The 550 MBit/s look extremely well compared to the 50 MBit/s that are achieved by a copying ORB and the standard TCP/IP stack. This means that a tenfold performance gain could have been achieved by introducing zero-copy through all the layers.

8.3 Parallelization Framework for CORBA

CORBA implements an inherent synchronous client/server paradigm (N to one) which has generally been considered unsuitable for parallel programming. The lack of peer-to-peer semantics and difficulty in achieving distributed concurrency and/or data flow prevented many applications. So what is needed is a parallelization paradigm (one to M) in addition to the (N to one) pattern.

None of the existing CORBA Services could match our requirements for a parallelizer service: neither asynchronous and decoupled calls, nor inherent load balancing, nor deliver at most once, nor dynamic publish & subscribe. We therefore designed and implemented the our own *Distributor Service* that could accommodate all these desired concepts and functionalities. In the evaluation the distributor service showed an impressive potential and performed very reliably.

With the implementation of a large part of a load balancing service inside the Distributor service a simple but flexible and portable service could be provided for MICO. The load balancer strategy which is responsible for dispatching the requests to the “right” server can be modified or exchanged on the fly.

The referential integrity of delivery suppliers and delivery consumers makes the service convenient for applications outside its dedicated target platform, the clusters of PCs. We evaluated the architecture for a small computational grid consisting of a large number of inhomogeneous computing resources spread all over our computer science department. The facilities of the service made it even possible to utilize some weak machines together as a respectable high performance computing resource by just adding this service to the CORBA world of applications.

The design of a distributed MPEG-4 Transcoder confined this and lead us to design a framework for parallel data processing with CORBA. The result is the modular, extensible, intuitive and simple to use *Distributor Framework*. This framework can be used to implement any distributed parallel application that deals with large amounts of data in a CORBA environment. The parallelization of jobs is achieved by using the *Distributor Service* which is the core component of the framework. The service schedules calls to different machines by an inherent load balancer and decouples the data supplier from the consumer.

The issue that triggered the development of this application and framework was the application of the zero-copy principle to middleware. As the framework deals with large amounts of data that has to be distributed to different processing nodes over the network zero-copy is essential to achieve a reasonable performance. First zero-copy is achieved by just using an optimized CORBA ORB and wrapping data into CORBA sequences of octets that can be communicated very efficiently. Second the ORB relies on a zero-copy TCP/IP stack that offers the standard socket API as its interface. Both prerequisites are justified and discussed by this thesis.

Another issue must be looked at, while discussing the efficiency and usage of the distributor service framework itself. Even if the service is transparent to the application implementor it must be well known that the calls including their parameters are sent to a service which forward them to a definite server. For large amounts of data such an indirection scenario would introduce much too much overhead. The distributor framework therefore exports the wrapped data as CORBA objects and sends a CORBA object reference as a parameter to the call. This is similar to a local processing through call by reference when just passing a pointer to a structure instead of copying the structure itself. Through the CORBA object reference the data is accessed directly from outside by the processing module that needs to access the data. This is a type of callback that is hidden by the CORBA middleware.

As a technology demonstrator an MPEG-Transcoder `trans` is based on this distributor framework and can therefore be implemented by just adding some domain specific modules or plugins to the framework. The overhead and complexity introduced by the parallelization and data distribution is completely hidden to the implementor. This is nothing new when working with objects and components but we also hide the overhead that generally would be introduced by network communication via middleware. We already showed a performance achievement of a factor of 10 for an optimized ORB communicating through a zero-copy operating system stack versus the original ORB. This tenfold performance gain can fully be used by our application.

The resulting tool is a command line based application for transcoding MPEG-2 DVD movies to the new MPEG-4 standard offering much better compression rates. The compute intensive encoding operation can be parallelized by just increasing the compute power by adding some more PCs to a cluster and starting another encoder task on these machines. The initialization, registering to a distributor service and the load balancing of jobs is adapted on run-time automatically. Like this real-time MPEG-4 encoding is enabled for full HDTV resolution or for multi-channel streams containing several parallel video streams.

A

MPEG-4 Compression – An Introduction

The following short introduction of the MPEG compression algorithms [89, 48, 18] is just intended to let the reader understand the field and is by no means complete.

MPEG-4 is a new standard for interactive multimedia creation, delivery, and playback for the Internet. MPEG-1 had its impact to the delivery of full-motion, full-screen video by CD-ROMs, MPEG-2 to the development of the DVD and MPEG-4 will have an impact on video over the Internet.

MPEG-4 is an extensive set of key enabling technology specifications with audio and video at its core. It was defined by the MPEG (Moving Picture Experts Group) committee, the working group within the International Organization for Standardization (ISO) that specified the widely adopted standards known as MPEG-1 and MPEG-2. MPEG-4, whose formal designation is ISO/IEC 14496, was finalized in October 1998 and became an international standard in early 1999.

Scenes and Media Objects

The central concept defined by the MPEG-4 standard is the audio-visual object, which is the foundation of the object-based representation. Such a representation is well suited for interactive applications and gives direct access to the scene contents.

A video object may consist of one or more layers to support scalable coding. The scalable syntax allows the reconstruction of video in a layered fashion starting from a standalone base layer, and adding a number of enhancement layers. This allows applications to generate a single MPEG-4 video bitstream for a variety of bandwidths and/or computational complexity requirements.

An MPEG-4 visual scene may consist of one or more video objects. Each video object is characterized by temporal and spatial information in the form of shape, motion, and texture. For certain applications video objects may not be desirable, because of either the associated overhead or the difficulty of generating video objects. For those applications, MPEG-4 video allows coding of rectangular frames which represent a degenerate case of an arbitrarily shaped object.

An MPEG-4 visual bitstream provides a hierarchical description of a visual scene as shown in Figure A.1. Each level of the hierarchy can be accessed in the bitstream by special code values called start codes. The hierarchical levels that describe the scene most directly are:

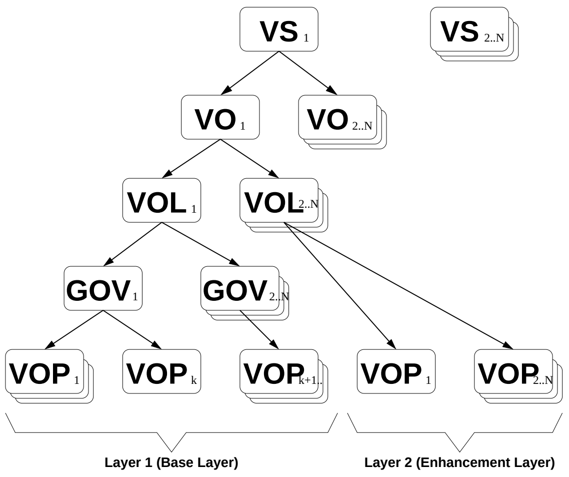


Figure A.1*: Logical structure of an MPEG-4 video bitstream*

Visual Object Sequence (VS): A container object combines the complete MPEG-4 scene which may contain any 2-D or 3-D natural or synthetic objects and their enhancement layers.

Video Object (VO): A video object corresponds to a particular (2-D) object in the scene. In the most simple case this can be a rectangular frame, or it can be an arbitrarily shaped object corresponding to an object or background of the scene.

Video Object Layer (VOL): Each video object can be encoded in scalable (multilayer) or non-scalable form (single layer), depending on the application, represented by the video object layer (VOL). The VOL provides support for scalable coding. A video object can be encoded using spatial or temporal scalability, going from coarse to fine resolution. Depending on parameters such as available bandwidth and computational power the desired resolution can be made available to the decoder.

Group of Video Object Planes (GOV): Each video object (VO) is sampled in time, each time sample of a video object is a video object plane (VOP). Video object planes can be grouped together to form a group of video object planes. GOVs can provide points in the bitstream where video object planes are encoded independently from each other, and can thus provide random access points into the bitstream.

Video Object Plane (VOP): A VOP is a time sample of a video object. VOPs can be encoded independently of each other, or dependent on each other by using motion compensation. A conventional video frame can be represented by a VOP with rectangular shape.

Motion Estimation and Error Compensation

Motion estimation and compensation are commonly used to compress video sequences by exploiting temporal redundancies between frames. The approaches for motion compensation in the MPEG-4 standard are similar to those used in other video coding standards. The main difference is that the block-based techniques used in the other standards have been adapted to the VOP structure used in MPEG-4. MPEG-4 provides three modes for encoding an input VOP, as shown in Figure A.2, namely:

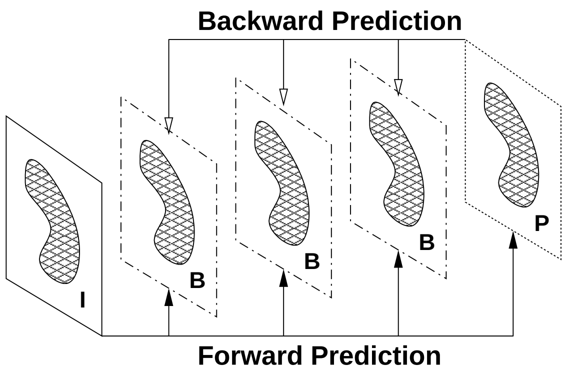


Figure A.2*: Dependency of the different I-, B- and P-planes used in video compression.*

I-Planes: A VOP may be encoded independently of any other VOP. In this case the encoded VOP is called an Intra VOP.

P-Planes: A VOP may be predicted (using motion compensation) based on another previously decoded VOP. Such VOPs are called Predicted VOPs.

B-Planes: A VOP may be predicted based on past as well as future VOPs. Such VOPs are called Bidirectional Interpolated VOPs. B-VOPs may only be interpolated based on I-VOPs or P-VOPs.

Texture Coding

The texture information of a video object plane is present in the luminance, Y, and two chrominance components, Cb, Cr, of the video signal. In the case of an I-VOP, the texture information resides directly in the luminance and chrominance components. In the

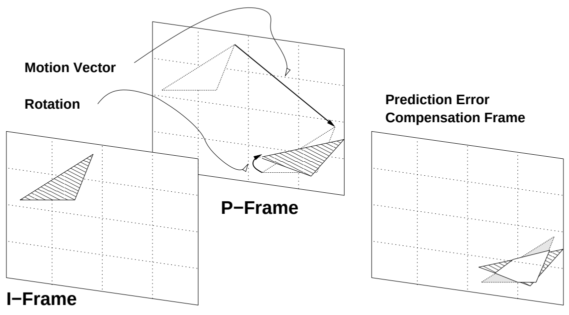


***Figure A.3**: Motion Vectors are calculated on the base of macro blocks. The rotation and pixel mutations are described by an error compensation matrix.*

case of motion compensated VOPs the texture information represents the residual error remaining after motion compensation (Figure A.3). For encoding the texture information, the standard 8x8 block-based DCT is used. To encode an arbitrarily shaped VOP, an 8x8 grid is superimposed on the VOP. Using this grid, 8x8 blocks that are internal to VOP are encoded without modifications. Blocks that straddle the VOP are called boundary blocks, and are treated differently from internal blocks. The transformed blocks are quantized, and individual coefficient prediction can be used from neighboring blocks to further reduce the entropy value of the coefficients. This is followed by a scanning of the coefficients, to reduce to average run length between to coded coefficients. Then, the coefficients are encoded by variable length encoding.

Discrete Cosine Transform (DCT) and Quantization

Internal video texture blocks and padded boundary blocks are encoded using a 2-D 8x8 block-based DCT. The DCT transform is followed by a quantization process.

The DCT coefficients are quantized as a lossy compression step. There are two types of quantizations available. Both are essentially a division of the coefficient by a quantization step size. The first method uses one of two available quantization matrices to modify the quantization step size depending on the spatial frequency of the coefficient. The second method uses the same quantization step size for all coefficients. MPEG-4 also allows for a non-linear quantization of DC values.

B

trans Manpage

NAME

trans - A fully scalable parallel and distributed MPEG2-to-MPEG4 video transcoder

SYNOPSIS

trans [-**i**<inMod>] [-**p**<proMod>] [-**e**<expMod>]
[-**w**<width>] [-**h**<height>] [-**r**<fps>] [-**b**<bitrate>] [-**f**<framesProJunk>]
[-**s**<stime>] [-**v**<verbosity>]
[- -**nofinchunks**<nofich>] [- -**nofoutchunks**<nofoch>]
[- -**inchannel**<ichname>] [- -**outchannel**<ochname>] [- -**strategy**<sty>]
[- -**pools**<nofp>]
[- -**forcerun**] [- -**disablecmd**] [- -**help**] <inFile> <outFile>

DESCRIPTION

trans allows the distributed transcoding of MPEG-2 streams or YV12-picture streams to MPEG-4 video streams. Based on CORBA the main task of the transcoding, the encoding itself, can be distributed to arbitrarily many components. Prerequisite is a running CORBA Distributor Service that is registered with the COS Naming Service.

The application is started as a command line tool that always needs its base configuration parameters. These consist mainly of the configuration of the functionality and import- and export-modules that an instance of **trans** shall use as well as the video parameters.

COMMAND LINE OPTIONS

Module description:

-i

Defines the import-module that shall be used. Available are an ImportChunk-module (c), an ImportFile-module (f) as well as an ImportGrabber-module that directly imports data from a frame grabber interface card. On importing chunks the CORBA Distributor Service must be available. (Default: f)

-p
Specifies the kind of chunk data processing. There is a ProcessNull-module (n) that just passes the data further to the export-modules, a ProcessMPEG4-module (e) that encodes the data with the help of an XviD library and a ProcessSorting-module (s) that sorts the single scenes. For DVD streaming there is an additional ProcessDVDDecode-module (d) that decodes MPEG-2 streams. The ProcessNull- and the ProcessSort-module need just one chunk pool. (Default: n)

-e
Defines the export of trans. Chunks can either be exported by the Distributor Service by using the ExportChunk-module (c), to one file with the ExportFile- (f) or to a file per chunk with the ExportMultiFile-module (m) and finally to an MP4 stream that already includes streaming information by the ExportMP4-module (s). (Default: f)

Video parameters:

-w
Defines the the width of the frame in pixel. (Default: 720)

-h
Defines the height of the frames in pixel. (Default: 480)

-r
Sets the frame rate in frames/s. (Default: 23.98)

General settings:

-s
Defines the time in seconds from the start of the module until the “Backup Loop” calls the scheduler for the first time (time to initialize all the datastructures and connections). (Default: 5)

-v
Verbosity of debug info. (Default: 0)

- -forcerun
Defines the time interval in seconds within that the “Backup Loop” calls the ’run’-method of the scheduler. (Default: 1)

- -disablecmd
The scheduler normally allows callbacks from the “Backup Loop” (e.g. run or stop). This behaviour can be disabled by this option. (Default: false)

Chunk-Pool parameters:

-f
Sets the the number of frames that are partitioned into one chunk. (Default: 25)

- -pools
Defines the number of Chunk Pools that are used, either a single pool (1) or and In- and and Out-pool (2). (Default: 2)

- -nofinchunks
Defines the number of default Chunks that shall be allocated in the In-Pool. This number has a large influence on the amount of memory that trans needs. (Default: 10)

- -nofoutchunks
Defines the number of default Chunks that shall be allocated in the Out-Pool. This number also has a large influence on the amount of memory that trans needs. (Default: 10)

Module specific parameters:

-b
Sets the bitrate of the MPEG4 encoder (in [KBit/s]). (Default: 500000)

- -inchannel
Defines the name of the Distributor Channel that the import-module uses. (Default: transcoderChannel)

- -outchannel
Distributor Channel that the export-module uses. (Default: transcoder-Channel). (Default: transcoderChannel)

- -strategy
Defines the strategy for a newly created Distributor Channel (In and Out) as there are Round-Robin (rr) or Least-Load (ll). (Default: rr)

- -help
Shows the help information.

Examples

The most simple call to **trans** can be used to just copy a file in a somehow a bit complicated way:

```
> trans inFile outFile
```

With the assistance of the MultiFile-export a YV12-stream can be partitioned into single frames. As there is no process-module needed (ProcessNull) one chunk pool is enough:

```
> trans -em -w720 -h480 -f1 --pools 1 \
  /tmp/starwars_720x480.yv12 /tmp/starwars
```

In the next example chunks of 25 frames each are exported to a Distributor Service. The encoder (or eventually many of them) directly accesses these exported chunks and encodes the data to MPEG4. After this the transformed data is just written to a file, e.g. in a network filesystem (Caution: The Distributor Service must be running and be registered with the Naming Service.):

```
> trans -ec -w720 -h480 -f25 --pools 1 \
  /tmp/starwars_720x480.yv12
> trans -ic -pe -em -w720 -h480 -f25 \
  - /tmp/starwars
```

To combine these MPEG-4 part sequences a second channel is used. To communicate over disjoint distributor channels the names of the particular channel must be indicated. We use the same example as above but the part streams are sent to a third component that sorts them. The ExportMP4-module concatenates the parts and packs the stream into an MP4 container file that can directly be streamed. The following commands implement exactely the scenario depicted in Figure 7.3:

```
> trans -ec -w720 -h480 -f25 --pools 1 \
  /tmp/starwars_720x480.yv12
> trans -ic -pe -ec -w720 -h480 -f25 \
  --outchannel toSortChannel - -
> trans -ic -ps -es -w720 -h480 -f25 --pools 1 \
  --inchannel toSortChannel - /tmp/starwars.mp4
```

C

IDL - Load Balancing Service

This IDL is based on the IDL from [150] and [76]. The following list summarizes the differences to these proposals:

- `ObjectId` corresponds to `PortableServer::ObjectID`
- The type `MemberId` was replaced with `ObjectId`
- Instead of a `LoadAlert`-Interface we use a `ProxyLoadMonitor`-Interface
- `Group`: `add_member` returns a `ProxyLoadMonitor` object
- `MemberData` is not needed anymore. Each strategy must manage its member data on its own

Group Factory Interface

The Group Factory manages the Load Balancing Groups. It mainly serves a generator of groups. Each group is identified by a unique ID by the factory context.

```
typedef string GroupId;
typedef sequence<GroupId> GroupIdSeq;

exception GroupExists {};
exception GroupNotFound {};

interface GroupFactory {
  Group       create(in GroupId id) raises(GroupExists);
  Group       get(in GroupId id) raises(GroupNotFound);
  GroupIdSeq  list();
};
```

ProxyLoadMonitor Interface

The Load Monitor component of a server keeps load information updated. On registrating with a group each member gets such a Load Monitor object which is used to submit load reports to the Load Balancing Service. For each re-subscribtion a new proxy is generated. The group therefore allows to temporarily lock a client.

```
typedef unsigned long     Load;
exception StrategyNotAdaptive {};

interface Strategy;          // Forward declaration

interface ProxyLoadMonitor {
  void push_load( in Load load ) raises ( StrategyNotAdaptive );
};
```

Group Interface

The group is the central element of the Load Balancing Service. Members register on a group and the group manages the balancing strategy.

```
typedef sequence<Object>  ObjectSeq;
typedef string            PropertyName;
typedef any               PropertyValue;
typedef CORBA::RepositoryId       RepId;
typedef PortableServer::ObjectId  ObjectId;
struct Property {
  PropertyName    name;
  PropertyValue   value;
};
typedef sequence<Property> PropertySeq;
struct PropertyError {
  PropertyName        name;
  PropertyErrorCode   code;
};
typedef sequence<PropertyError> PropertyErrorSeq;
enum PropertyErrorCode { BAD_PROPERTY, BAD_TYPE, BAD_VALUE };

exception MemberExists {};
exception MemberNotFound {};
exception StrategyNotFound {};
exception NoMembersReady {};
exception NoMembers {};
exception InvalidProperties {
  PropertyErrorSeq error;
};

interface Strategy ;         // Forward declaration
interface CustomStrategy ;   // Forward declaration

interface Group {
  GroupId id();

  ProxyLoadMonitor add_member( in Object id )
    raises( MemberExists );
  void remove_member( in Object id ) raises( MemberNotFound );
  ObjectSeq list() raises( NoMembers );
  void lock_member( in Object id )
    raises( MemberNotFound, StrategyNotFound );
  void unlock_member( in Object id )
    raises( MemberNotFound, StrategyNotFound );

  void set_builtin_strategy(in string name, in PropertySeq props)
    raises( StrategyNotFound, InvalidProperties);
  void set_custom_strategy(in CustomStrategy s);
  Strategy get_strategy() raises( StrategyNotFound );

  Object get_server()
    raises( StrategyNotFound, NoMembersReady, NoMembers );
};
```

Strategy Interface

The Strategy-interface implements the reception of load reports of a group member. This load reports allows a fair load balancing.

```
interface Strategy {
  string name();
  PropertySeq get_properties();
  void set_properties( in PropertySeq props )
    raises( InvalidProperties );
  void push_load( in Object id, in Load load )
    raises( StrategyNotAdaptive );
};
```

CustomStrategy Interface

The Load Balancing Service offers the user to implement its own load balancing strategy. To use such a strategy that implements the `Strategy`-interface some administration functionality must be provided by them. These are afterwards accessed and used by the load balancing group.

```
interface CustomStrategy : Strategy {
  Object      next() raises( NoMembers, NoMembersReady );
  void        set_members( in ObjectSeq members );
  void        lock_member( in Object id );
  void        unlock_member( in Object id );
};
```

D

IDL - Distributor Service

As our new CORBA Distributor Service is an extention of the COS Event Service [110], its interface is very much alike the COS Event Service interface. The main differences are listed in the following points:

- All "EventChannel"-designations were changed to "DistChannel"
- All "Pull"-methods were withdrawn
- The Load Balancing IDL definition from Appendix C is imported into the Distributor Service IDL.
- The `DistChannelFactory`-interface newly possesses a second "create"-method: `create_distchannel_with_group`.
- The distributor channel additionally provides an `ChannelAdmin`-interface.
- The `ProxyPushSupplier`-interface has a new method `getProxyLoad-Monitor`.

Distributor Communication Module

The **`DistComm`**-module contains the Consumer- and Supplier- Interfaces.

```
module DistComm {

   exception Disconnected {};
   exception NoConsumers {};
   exception NoConsumersReady {};

};
```

Push Consumer Interface

The **`PushConsumer`**-interface contains all methods, which a Supplier can call. The **`ProxyPushConsumer`**-interface is derived from the **`PushConsumer`**-interface.

```
module DistComm {

   interface PushConsumer {
      void push (in any data)
          raises(Disconnected, NoConsumers, NoConsumersReady );
      void disconnect_push_consumer();
   };

};
```

Push Supplier Interface

The **`PushSupplier`**-interface includes all the methods that a consumer can call. The **`ProxyPushSupplier`**-interface is derived from the **`PushSupplier`**-interface.

```
module DistComm {

   interface PushSupplier {
      void disconnect_push_supplier();
   };

};
```

Distributor Channel Administration Module

The `DistChannelAdmin`-interface contains the objects implementing the main functionality of the Distributor Service, the Distributor Channels. But it also implements the Interface for administering these channels and provides the Proxy-Interface definitions. Some of these interfaces need the definition of the Load Balancing IDL (Appendix C).

```
#include ``CosLoadBalancing.idl'';

module DistChannelAdmin {

   exception AlreadyConnected {};

};
```

Proxy Push Consumer Interface

This interface represents the Consumer-interface for the suppliers.

```
module DistChannelAdmin {

   interface ProxyPushConsumer: DistComm::PushConsumer {
      void connect_push_supplier( in DistComm::PushSupplier
                                     push_supplier)
              raises(AlreadyConnected);
   };

};
```

Proxy Push Supplier Interface

Analogous this interface represents the Supplier-Interface for the consumers.

```
module DistChannelAdmin {

   interface ProxyPushSupplier: DistComm::PushSupplier {
      void connect_push_consumer( in DistComm::PushConsumer
                                     push_consumer)
         raises(AlreadyConnected);
      CosLoadBalancing::ProxyLoadMonitor getProxyLoadMonitor();
  };

};
```

Consumer Administration Interface

This interface can be accessed through the Distributor Channel. It is only intended for delivery consumers.

```
module DistChannelAdmin {

   interface ConsumerAdmin {
      ProxyPushSupplier obtain_push_supplier();
   };

};
```

Supplier Administration Interface

This interface can also be accessed through the Distributor Channel. It is only intended for delivery suppliers.

```
module DistChannelAdmin {

   interface SupplierAdmin {
      ProxyPushConsumer obtain_push_consumer();
   };

};
```

Channel Administration Interface

This interface can be accessed through the Distributor Channel and serves for channel administration. It can mainly define the load balancing strategies and settings.

All the methods are independent from the implementation of the used Load Balancing Group. If the Load Balancing Service of Appendix C is used (default), the usage of this method is the same as specified there.

```
module DistChannelAdmin {

   interface ChannelAdmin {
      void set_builtin_strategy(in string name,
                       in CosLoadBalancing::PropertySeq props)
         raises( CosLoadBalancing::StrategyNotFound,
                       CosLoadBalancing::InvalidProperties);
      void set_custom_strategy(
                       in CosLoadBalancing::CustomStrategy s );
      CosLoadBalancing::Strategy get_strategy()
         raises( CosLoadBalancing::StrategyNotFound );
  };

};
```

Distributor Channel Interface

The Distributor Channel does not implement functionality that can be accessed from outside.

```
module DistChannelAdmin {

   interface DistChannel {
      ConsumerAdmin for_consumers();
      SupplierAdmin for_suppliers();
      ChannelAdmin  for_admin();
      void destroy();
   };

};
```

SimpleDistChannelAdmin Modul

The `SimpleDistChannelAdmin`-module contains the `Distributor-Channel-Factory`.

```
#include ``LoadBalancing.idl'';

module SimpleDistChannelAdmin {

   exception GroupNotEmpty {};

};
```

Distributor Channel Factory Interface

The `Distributor-Channel-Factory` implements the methods used to create new channels. A new channel can be initialized by an optional Load Balancing Group.

```
module SimpleDistChannelAdmin {

   interface DistChannelFactory {
      DistChannelAdmin::DistChannel create_distchannel ();
      DistChannelAdmin::DistChannel
           create_distchannel_with_group(
                              in CosLoadBalancing::Group group )
         raises( GroupNotEmpty );
   };

};
```

Bibliography

[1] IEEE Std 1596-1992. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE Computer Society, August 1993.

[2] 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.

[3] 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.

[4] Inc. Alteon WebSystems. Jumbo frames. http://www.alteon.com/products/jumbo_frames.html.

[5] C. Amza, A. Cox, S. Dwarkadas, C. Hyams, Z. Li, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feb 1996.

[6] D. P. Anderson and D. Ferrari. The DASH Project: An Overview. Technical Report 88/405, CS Div., EECS Dept., UC Berkeley, February 1988.

[7] E. W. Anderson and J. Pasquale. The performance of the container shipping i/o system. In *Symposium on Operating Systems Principles*, page 229, 1995.

[8] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. ”a case for networks of workstations: Now”. *IEEE Micro*, Feb 1996.

[9] Scali AS. *ScaMPI User's Guide*, 1997. http://www.scali.com/html/scampi.html.

[10] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. NAS Parallel Benchmark. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[11] D. Becker and P. Merkey. The Beowulf Project. http://www.beowulf.org.

[12] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of 1995 ICPP Workshop on Challenges for Parallel Processing*, Oconomowc, Wisconsin, U.S.A., August 1995. CRC Press.

[13] P. A. Bernstein. Middleware: A Model for Distributed Services. *Communications of the ACM*, 39(2):86–97, February 1996.

[14] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[15] A. D. Birrel and B. J. Nelson. Implementing remote procedure calls. *ACM Transaction on Computer Systems*, 2(1):39–59, February 1984.

[16] N. J. Boden, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet - A Gigabit per Second Local Area Network. In *IEEE Micro*, volume 15(1), pages 29–36, February 1995.

[17] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Pub Co, 2nd edition edition, February 1994.

[18] F. Bossen and T. Ebrahimi. A simple and efficient binary shape coding technique based on bitmap representation. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'97)*, volume 4, pages 3129–3132, Munich, Germany, April 1997.

[19] N. Brown and C. Kindel. Distributed Component Object Model Protocol – DCOM/1.0. Technical report, Microsoft Corporation, 1996.

[20] J. Brustoloni and P. Steenkiste. Copy emulation in checksummed, multiple-packet communication. In *Proceedings of IEEE INFOCOM 1997*, pages 1124–1132, Kobe, Japan, April 1997.

[21] J. C. Brustoloni. *Effects of Data Passing Semantics and Operating Systems Structure on Network I/O Performance*. PhD thesis, School of Computer Science, Carnegie Mellon, Sept 1997. Published as CMU Tech Report CMU-CS-97-176.

[22] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. 2nd Symp. on Operating Systems Design and Implementation (OSDI)*, pages 277–291, Seattle, WA, Oct 1996. USENIX.

[23] D. R. Butenhof. *Programming With POSIX Threads*. Addison-Wesley, 1997.

[24] K. Cameron, L. J. Clarke, and A. G. Smith. CRI/EPCC MPI for T3D. In *Conference paper, 1st European Cray T3D Workshop, EPFL*, Sept 1995. http://www.epcc.ed.ac.uk/t3dmpi/Product/Performance/index.html.

[25] J. B. Carter and W. Zwaenepoel. Optimistic Implementation of Bulk Data Transfer Protocols. In *Proceedings of the International Conference on Measurement*

and Modeling of Computer Systems, Sigmetrics, pages 61–69, Berkeley, CA, May 1989.

[26] J. Chase, D. Anderson, A. Gallatin, A. Lebeck, and K. Yocum. Network I/O with Trapeze. In *Proceedings of Hot Interconnects Symposium*, Stanford, CA, August 1999.

[27] J. Chase, A. Gallatin, and K. Yocum. End System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[28] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 253–264, San Diego, CA, USA, Jan 1996. The USENIX Association.

[29] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, Jun 1989.

[30] G. Coulson and S. Baichoo. A Configurable Multimedia Middleware Platform. *IEEE Multimedia*, 6(1):62–76, Jan 1999.

[31] G. Coulson and S. Baichoo. Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment. *ACM Distributed Computing Journal*, 14(2):113–126, April 2001. Springer Verlag Press.

[32] P. J. Courtois. Decomposability—queueing and computer system applications, 1977.

[33] P. J. Courtois. On time and space decomposition of complex structures. *Communications of the ACM*, 28(6), June 1985.

[34] Cray Research Inc. *CRAY T3D Applications Programming Course and Cray T3D Hardware Reference Manual*, Nov 1993. TR-T3DAPPL.

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

[36] J. D. Day and H. Zimmerman. The OSI Reference Model. In *Proc of the IEEE*, volume 71, pages 1334–1340, Dec 1983.

[37] E. W. Dijkstra. The structure of the THE multi-programming system. *Communications of the ACM*, 11 1968.

[38] E. W. Dijkstra. Complexity controlled by hierarchical ordering of function and variability. In N. Naur and B. Randell, editors, *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, pages 181–185. NATO Scientific Affairs Division, JAN 1969.

[39] E. W. Dijkstra. Notes on structured programming. In *Structured Programming*. Academic Press, 1969.

[40] E. W. Dijkstra. Hierarchial ordering of sequential processes. *Acta Informatica, Springer Verlag (Heidelberg, FRG and NewYork NY, USA) Verlag*, 1(2), OCT 1971.

[41] Dolphin Interconnect Solutions. *PCI SCI Cluster Adapter Specification*, 1996.

[42] J. Dongarra, R. Hemoel, A. Hey, and D. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, ORNL, 1993.

[43] P. Druschel. *Operating System Support for High-speed Networking*. PhD thesis, University of Arizona, Aug 1994.

[44] P. Druschel and L. L. Peterson. FBufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 189–202, Asheville, NC, Dec 1993.

[45] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. Vmmc-2: Efficient support for reliable, connection-oriented communication. In *In Proceeding of Hot Interconnects V*, August 1997.

[46] C. Dubnicki, E. W. Felten, L. Iftode, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proc. 10th Intl. Parallel Prof. Symp.*, pages 372–381, Honolulu, HI, April 1996. IEEE.

[47] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, March-April 1998.

[48] T. Ebrahimi and C. Horne. MPEG-4 Natural Video Coding - An Overview. Technical report, Swiss Federal Institute of Technology EPFL, 1999.

[49] K. Fall. *A Peer-to-Peer I/O System in Support of I/O Intensive Workloads*. PhD thesis, University of California, San Diego, 1994.

[50] K. Fall and J. Pasquale. Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability. In *Proc. of the USENIX Winter Technical Conference*, pages 327–334, San Diego, CA, Jan 1993.

[51] P. Felber, R. Guerraoui, and R. Guerraoui. Towards Reliable CORBA: Integration vs. Service Approach. In Max Mühlhäuser, editor, *Workshop Reader of the 10th European Conference on Object-Oriented Programming (ECOOP'96*, pages 199–205, Linz, Austria, 1997. dpunkt Verlag.

[52] P. Felber, R. Guerraoui, and A. Schiper. Replicating Objects using the CORBA Event Service. In *Proc. of the 6th Workshop on Future Trends of Distributed Computing Systems (FTDCS'097)*, pages 14–19. IEEE, Oct 1997.

[53] P. Felber, R. Guerraoui, and A. Schiper. The CORBA Object Group Service, 1997.

[54] P. Felber, R. Guerraoui, and A. Schiper. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[55] J. Floyd, S.and Mahdavi, M. Mathis, , and M. Podolsky. An extension to the Selective Acknowledgement (SACK) Option for TCP, July 2000. RFC 2883, Proposed Standard.

[56] S. Floyd. Congestion Control Principles, Sept 2000. RFC 2914.

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

[58] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org, July 1997.

[59] The OpenMP Forum. OpenMP C and C++ Application Program Interface, Version 1.0. http://www.openmp.org, October 1998.

[60] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1st edition, November 1998. ISBN: 1558604758.

[61] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In *Proceedings of Annual USENIX Technical Conference*, Monterey, CA, June 1999. USENIX.

[62] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Object-Oriented Software*. Addison Wesley, 1995.

[63] P. Geoffray. OPIOM: Off-Processor IO with Myrinet. In *Proc. of 1st International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 261–268, Brisbane, Australia, May 2001. IEEE.

[64] P. Geoffray, L. Prylli, and B. Tourancheau. BIP-SMP: High performance message passing over a cluster of commodity SMPs. In *In Proceedings of Super Computing (SC99)*, Portland, USA, Nov 1999.

[65] 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.

[66] 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.

[67] T. Gross and D. O'Hallaron. *iWarp: Anatomy of a Parallel Computing System*. MIT Press, 1998.

[68] Distributed Systems Research Group. CORBA Comparison Project Extension, Final Report. http://www.omg.org, June 1999.

[69] Moving Pictures Expert Group. MPEG-2 Standard, 1998-2000. ISO/IEC 13818:1-10.

[70] Moving Pictures Expert Group. MPEG-4 Standard, 1998-2000. ISO/IEC 14496.

[71] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, San Mateo, California, second edition, 1995.

[72] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman, Inc., 1999.

[73] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, December 2000.

[74] Apple Inc. Darwin Streaming Server: Open Source QuickTime Streaming Server. http://developer.apple.com/darwin/projects/streaming/.

[75] GigaNet Inc. http://www.giganet.com/.

[76] Tri-Pacific Software Inc. and VERTEL Corporation. Load Balancing and Monitoring. http://www.omg.org/, 2001. orbos/2001-08-01.

[77] American National Standards Institute. *ANSI NCITS 337-2000, Information Technology - Scheduled Transfer (ST)*. ANSI, Washington DC, 2000.

[78] Information Sciences Institute. DARPA Internet Program Protocol Specification NIC RFC 791. In *DDN Protocol Handbook*, volume 2, pages 99–149. , Sep 1981.

[79] Information Sciences Institute. Transmission Control Protocol NIC RFC 793. In *DDN Protocol Handbook*, volume 2, pages 179–198. , Sep 1981.

[80] InterProphet. SiliconTCP^{TM}:a new way to do internet communications. http://www.interprophet.com/, http://www.ethersan.com/.

[81] IONA and Isis. An Introduction to Orbix+Isis. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[82] ITA. *InfiniBand Architecture Specification Volume 1.0a*. Infiniband Trade Association, 1.0a edition, June 2001.

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

[84] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance RFC 1323. In *DDN Protocol Handbook*. , May 1992.

[85] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proc. ACM Communications Architectures and Protocols Conf. (SIGCOMM)*, pages 259–268, San Francisco, CA, Sep 1993.

[86] 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.

[87] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. 1994 Winter Conf.*, pages 115–131. USENIX, Jan 1994.

[88] C. A. Kent and J. G. Mogul. Fragmentation considered harmful. *WRL Technical Report 87/3*, 1987.

[89] R. Koenen. Overview of the MPEG-4 Standard. Overview N3156, Moving Pictures Expert Group, 1994. Maui Version.

[90] H. T. Kung, R. Sansom, S. Schlick, P. Steenkiste, M. Arnould, F. Bitz, F. Christianson, E. Cooper, O. Menzilcioglu, D. Ombres, and B. Zill. Network-Based Multicomputers: An Emerging Parallel Architecture. In *Proc. Supercomputing '91*, pages 664–673, Albuquerque, NM, Nov 1991. IEEE.

[91] Ch. Kurmann, F. Rauch, and T. Stricker. Speculative Defragmentation - Leading Gigabit Ethernet to True Zero-Copy Communication. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 4(1):7–18, March 2001. ”Short version available in Proceedings of the 9th International Symposium on High Performance Distributed Computering HPDC, Pittsburgh, Pennsylvania, August 1-4, 2000”.

[92] Ch. Kurmann and T. Stricker. Memory System Performance Characterization with ECT memperf - Extended Copy Transfer Characterization. http://www.cs.inf.ethz.ch/CoPs/ECT/.

[93] Ch. Kurmann and T. Stricker. Characterizing memory system performance for local and remote accesses in high end SMPs, low end SMPs and clusters of SMPs. In *Proc. of 7th Workshop on Scalable Memory Multiprocessors held in conjunction with ISCA98*, Barcelona, Spain, June 1998. IEEE.

[94] S. Leffler, K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison Wesley, 1989.

[95] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321 – 359, November 1989.

[96] 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.

[97] S. Lo and S. Pope. The Implementation of a High Performance ORB over Multiple

Network Transports. In *Proceedings of the Middleware Conference '98*, pages 157–172, The Lake District, England, Sept 1998.

[98] D. Mackie and B. May. MPEG4IP: Open Source, Open Standards, Open Streaming. http://mpeg4ip.sourceforge.net/.

[99] S. Maffeis. Electra—Making Distributed Programs Object-Oriented. In *Proc. of the Usenix Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 143–156, San Diego, CA, 1993.

[100] S. Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the Conference on Object-Oriented Technologies*, pages 135–146, Monterey, CA, June 1995. USENIX.

[101] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, Switzerland, Feb 1995.

[102] J. D. McCalpin. Sustainable memory bandwidth in current high performance computers. Technical report, University of Delaware, 1995.

[103] F. W. Miller, P. Keleher, and S. K. Tripathi. General Data Streaming. In *Proc. 19th IEEE Real-Time Systems Symposium*, pages 232–41, Madrid, Dec 1998. IEEE.

[104] R. Numrich, P. Springer, and J. Peterson. Measurement of Communication Rates on the Cray T3D Interprocessor Network. In *Proc. HPCN Europe '94, Vol. II*, pages 150–157, Munich, April 1994. Springer Verlag. Lecture Notes in Computer Science, Vol. 797.

[105] S. W. O'Malley, M. B. Abbot, N. C. Hutchinson, and L. L. Peterson. A Transparent Blast Facility. *Internetworking: Research and Experience*, 1(2), Dec 1990.

[106] OMG. Discussion of the Object Management Architecture (OMA) Guide. http://www.omg.org, 1997.

[107] OMG. Asynchronous Messaging. http://www.omg.org, May 1998. orbos/98-05-05.

[108] OMG. CORBAservices: Common Object Services Specifiaction. http://www.omg.org, Dez 1998. formal/98-12-09.

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

[110] OMG. Event Service Specification. http://www.omg.org/, 2000. formal/00-06-15.

[111] OMG. Notification Service Specification. http://www.omg.org/, 2000. formal/00-06-20.

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

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

[114] OMG. CORBA 3 Full Specification. http://www.omg.org/, 2002. formal/02-06-33.

[115] OMG. Fault Tolerant CORBA, Chapter 23 CORBA 3 Specification. http://www.omg.org, Sept 2002. formal/02-06-33.

[116] OMG. The Portable Object Adapter, Chapter 11 CORBA Specification. http://www.omg.org, Sept 2002. formal/02-06-33.

[117] T. Östreich. transcode: Linux Video Stream Processing Tool. http://www.theorie.physik.uni-goettingen.de/ ostreich/transcode/.

[118] O. Othman, C. O'Ryan, and D. Schmidt. The Design of an Adaptive CORBA Load Balancing Service. *IEEE Distributed Systems Online*, 2, Apr 2001.

[119] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 15–28, 1999.

[120] M. Pakin, S. Buchanan, M. Lauria, and A. Chien. The Fast Messages (FM) 2.0 streaming interface. In *Proc. USENIX '97*, 1997.

[121] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2):60–73, 1997.

[122] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. of the 1995 ACM/IEEE Supercomputing Conference*, San Diego, CA, Dec 1995.

[123] D. L. Parnas. On a buzzword: Hierarchical structure. In *Proc. of IFIP Congr. 1974*, pages 336–339, Amsterdam, The Nederland, 1974. North-Holland Publ.

[124] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. *IEEE MICRO*, 22(1):46–57, February 2002.

[125] P. Pietikainen. Hardware-Assisted Networking Using Scheduled Transfer Protocol On Linux. Master's thesis, University of Oulu, Finland, 2001.

[126] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proceedings of IEEE INFOCOM '01*, pages 67–76, Anchorage, Alaska, USA, April 2001.

[127] 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.

[128] The Gnome Project. ORBit: A CORBA 2.2-compliant Object Request Broker. http://orbit-resource.sourceforge.net/.

[129] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. Technical report, LHPC and INRIA ReMaP, ENS-Lyon, 1997. http://lhpca.univ-lyon1.fr/.

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

[131] F. Rauch, Ch. Kurmann, and T. Stricker. Partition Cast — Modelling and Optimizing the Distribution of Large Data Sets on PC Clusters. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Lecture Notes in Computer Science 1900, Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference Munich*, Munich, Germany, Aug 2000. Springer. Also available as Technical Report 343, Department of Computer Science, ETH Zürich, http://www.inf.ethz.ch/.

[132] F. Rauch, Ch. Kurmann, and T. Stricker. Optimizing the Distribution of Large Data Sets in Theory and Practice. *Concurrency and Computation: Practice and Experience*, 14(3):165–181, April 2002. John Wiley & Sons, Ltd.

[133] W. Ruh, T. Herron, and P. Klinker. *IIOP Complete, Understanding CORBA and Middleware Interoperability*. Addison-Wesley, Oct 1999. ISBN 0-201-37925-2.

[134] 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.

[135] 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.

[136] R. Seifert. *Gigabit Ethernet: Technology and Applications for High-Speed LANs*. Addison-Wesley, May 1998. ISBN: 0201185539.

[137] P Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of Supercomputing Conference SC2001*, Denver, Colorado, USA, November 2001. IEEE Computer Society.

[138] H. A. Simon and A. Ando. Aggregation of variables in dynamic systems. In *Econometrica 29*, pages 876–893, 1961.

[139] T. Sterling, L. Salmon, D. J. Becker, and D. F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, May 1999.

[140] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2l: Software coherent shared memory

on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–183, October 1997.

[141] W. R. Stevens. *TCP/IP Illustrated, Vol. 1 and Vol. 2.* Addison Wesley, 1993.

[142] T. Stricker and T. Gross. Optimizing Memory System Performance for Communication in Parallel Computers. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 308–319, Portofino, Italy, June 1995. ACM/IEEE.

[143] T. Stricker and T. Gross. Global Address Space, Non-Uniform Bandwidth: A Memory System Performance Characterization of Parallel Systems. In *Proceedings of the ACM conference on High Performance Computer Architecture (HPCA3)*, 1997.

[144] 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.

[145] 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-96-166, CMU-CS-00-133 (REV).

[146] Inc. Sun Microsystems. JavaTM RMI over IIOP. http://java.sun.com/products/rmi-iiop/.

[147] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[148] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation*. Prentice-Hall, Inc., second edition, 1999.

[149] M. Taufer. Personal communication. ETH Zurich, Sept 2000.

[150] IONA Technologies. Load Balancing RFP. http://www.omg.org/, 2001. orbos/2001-08-05.

[151] H. Tezuka, A. Hori, and Y. Ishikawa. PM: A High-Performance Communication Library for Multi-user Parallel Environments. Technical Report TR-96015, RWC, Real World Computing Partnership, 1996.

[152] The OpenMP Forum. OpenMP Fortran Application Program Interface, Version 1.0. http://www.openmp.org, October 1997.

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

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

[155] The Specification for the Virutal Interface Architecture. http://www.viarch.org/.

[156] T. von Eicken, A. Basu, and V Buch. Low Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, 15(1):46–53, Feb 1995.

[157] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of 15th Symposium on Operating Systems Principles (SOSP-15)*, Cooper Mountain, CO, USA, Dec 1995. ACM.

[158] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. 19th Intl. Conf. on Computer Architecture (ISCA)*, pages 256–266, May 1992.

[159] P. Wegner. Concepts and paradigms of object-oriented programming. *ACM SIGPLAN OOPS Messenger*, 1(1):7–87, 1990.

[160] M. Welsh, A. Basu, and T. von Eicken. Low-Latency Communication over Fast Ethernet. In *Proceedings of EuroPar '96*, pages 187–194, Lyon, France, August 1996.

[161] M. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *In Proceedings of High Performance Computer Architecture HPCA 3*, San Antonio, TX, February 1997.

[162] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. Technical Report TR97-1620, Cornell University, August 1997.

[163] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[164] XviD: Open Source ISO MPEG-4 Video Codec. http://www.xvid.org/.

[165] W. Zwaenepoel. Protocols for large data transfers over local area networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22–32, Whistler Moutain, British Columbia, Canada, 1985.

Curriculum Vitae

Christian Kurmann

December 5, 1970	Born in Luzern, Switzerland Citizen of Switzerland Son of Alois and Heidi Kurmann
1977–1986	Primary and Secundary School Gossau SG, Switzerland
1986–1991	State College, Kantonsschule St.Gallen St.Gallen, Switzerland
1991	Matura/Baccalaureate Typus C, Kantonsschule St.Gallen
1991–1996	Studies in Computer Science Swiss Federal Institute of Technology ETH Zurich, Switzerland
1993–1994	Internship at Control Design and Development Ltd Peterborough, England
1996	Masters Diploma in Computer Science and Engineering, ETH Zurich, Switzerland Diploma thesis on “Compression Domain Volume Rendering for Distributed Environments”
1996–2002	Research and teaching assistant in the Parallel and Distributed Systems Group headed by Prof. Thomas M. Stricker, Institute for Computer Systems, Swiss Federal Institute of Technology ETH, Zurich, Switzerland