Innovative and Efficient Communication Methods for System Area Networks

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Dipl. Inf. Markus Fischer
aus Holzminden

Mannheim, 2002

Dekan:	Professor Dr. Herbert Popp, Universität Mannheim
Referent:	Professor Dr. Ulrich Brüning, Universität Mannheim
Korreferent:	Professor Dr. Volker Lindenstruth, Universität Heidelberg

Tag der mündlichen Prüfung: 5. Dezember 2002

Acknowledgments

The results of this thesis have been achieved over the recent years. There are quite a few people which I have enjoyed working with and therefore I would like to thank.

I would like to thank the people from the RWCP project, in particular Y. Ishikawa-san, A. Hori-san and J. Nolte-san.

I also would like to thank the PVM team at ORNL/UTK. Especially Bob Manchek for his insight into portable software for heterogeneous distributed environments.

Special thanks go to the people from Myricom for their overall support.

This thesis however would not have been possible without the support and guidance from my advisor Professor Dr. Ulrich Brüning. He was and still is a great resource for any kind of technical question and I enjoyed in particular how easy things looked after listening to his explanations. I also have to be grateful on his valuable comments while writing this thesis.

Last but not least, I have to thank those who supported and motivated me over an extended amount of time.

Finally, this can not be a complete list, but whoever reads this but is not mentioned in particular, will by herself or himself know whether she or he is missing here ;-)

Abstract

A new trend is emerging to replace massively parallel machines with clusters built from Commercial Off The Shelf (COTS) components. Clusters typically consist of standard compute nodes and an interconnection network. In order to achieve high efficiency, a parallel application is very dependent on the communication system. Traditional communication interfaces will not let an application exploit the given resources due to overburdened protocols. System Area Networks (SAN) have been developed with the main purpose of supporting user-level communication (ULC) systems. User level communcation bypasses the operating system from the critical communication path. One aspect, however, still needs to be solved: How applications will benefit directly from available resources and in which way protocols can be developed to directly support existing applications.

This is one of the reasons why traditional networks using standard protocol stacks remain preferred, thus offering upgrade compatibility.

This thesis addresses existing communication principles and provides new protocols for efficient communication. It also provides message passing protocols for the new Atomic Low Latency (ATOLL) System Area Network. This network on a chip solution was analyzed and an extended design was developed which will enhance the current implementation to enable protocol offloading for better resource utilization. Finally, this thesis will present a new middleware layer, which will enable the replacement of traditional networks by providing compatible protocols at binary level. This middleware layer will offer the same semantics at much higher performance. As will be shown, standard existing applications will experience much better performance with an improvement in the order of a magnitude.

Contents

List of Figures

Chapter 1

Introduction

During the past few years, the platform for parallel and high performance computing has been changing. A major shift from massively parallel systems to clusters of workstations and Personal Computers (PCs) can be seen. The latter are built up from commercial off the shelf (COTS) components, which partly offer high performance known only from supercomputers, however at a fraction of the cost. This is especially true for the central processing unit (CPU). Todays standard CPUs offer a price performance ratio which does not make it attractive to develop a new CPU for high performance computing.

When using standard CPUs, the host system itself offers very limited possibilities for own extensions. For example the host system bus, the memory interface, the I/O subsystem are all proprietary developments and do not offer any chances for compelling research.

However, the communication subsystem which plugs into the I/O subsystem as a network interface card, for example, offers the possibility for individual development. Currently the Peripheral Component Interconnect (PCI) bus is the I/O standard for commodity systems and is supported by any operating system offering a very portable solution for developing communication systems.

There exist standard Ethernet type network interfaces, which are used to connect to the Internet. However, communication intensive applications are hampered by standard networks such as Fast Ethernet with its overburdened TCP/IP protocol. In particular, the high latency for small messages, which is in the range of several hundreds of microseconds, as well as the low bandwidth for bulk messages, which peaks at about 8-9MBytes/s when using Fast Ethernet, slow down parallel applications.

Within a cluster, the distance between nodes is rather small and hardware error rates are extremely low. Thus, a light weight protocol for message transfer is more practical.

Since several years, new high speed networking devices exist to build up a system area network (SAN) , which delivers performance on message transfers in the range of Gigabits/s. Today, popular high speed interconnects are Myrinet, Quadrics, GigaNet, Scalable Coherent Interface (SCI) and Servernet, which are all available with an interface into the PCI bus.

Currently a high percentage of existing clusters is still equipped with standard network devices such as Fast Ethernet. As of today, the TopClusters web site [74] which lists the largest unclassified installations of clusters, shows that 120 out of 200 (60%) installations are equipped with an Ethernet network. This is mainly for compatibility reasons since applications based on the standardized TCP/IP are easily portable. This protocol however is known to cause too much overhead [34]. Lowering latency is an important key to achieve good communication performance. A survey on message sizes shows that protocols and hardware have to be designed to handle short messages extremely well [118]:

- in seven parallel scientific applications 30% of the messages were between 16 bytes and a kilobyte
- the median message sizes for TCP and UDP traffic in a departmental network were 32 and 128 bytes respectively
- 99% of TCP and 86% of the UDP traffic was less than 200 bytes
- on a commercial database all messages were less than 200 bytes
- the average message size ranges between 19 - 230 bytes

Recent research on Gigabit/s interconnects has shown that one key to achieve low latency and high bandwidth is to bypass the operating system, avoiding a trap into the operating system: User Level Communication (ULC) gives the user application full control over the interconnect device. Some of the initiating projects which focused on lowering the overhead for communication are for example HPVM, UNET, Active Messages.

In addition to the replacement of proprietary hardware, there exists a trend to replace proprietary operating systems with standard open source software: LINUX is an open source operating system under the GNU Public

License (GPL), which has become a very popular platform allowing the development of kernel extensions which can be useful when integrating additional devices. Currently it is used in over 90% of the largest cluster installations [74].

1.1 Evolution of Networking

This section will describe in which way several network implementations differ in terms of hardware and software. It will also analyze why clusters are equipped with a high speed network.

1.1.1 Local- and Wide Area Networks

A short introduction on the functionality of Local- and Wide Area Networks will be given in this section. Although clusters (or network of workstations [94]) are preferably equipped with high speed interconnects, a large fraction still use Fast Ethernet (see listing of clusters in [74] for example). Fast Ethernet however is used to implement Local- (LAN) and Wide Area Networks (WAN) in which data is transfered through gateways and routers over longer distances. Figure 1.1 depicts such an environment which forms the Internet with a myriad of installations. Every added system is not joining the Internet, but is actually becoming a part of it.

Given the widespread use of Fast Ethernet network cards, they practically come with no additional cost. Being in use over the years, any system process which is associated with network transactions will use the Ethernet protocol for communication. This way it is possible to establish wide area connections as well, since the communicating endpoints will not be able to determine their distance. Routers and gateways will forward traffic to the final destination transparently. In order to achieve this, the Ethernet protocol must be very fault tolerant. Several checksums for data integrity and additional control flow will implement a reliable protocol. At the physical level a performance of 100Mbit/s was introduced with Fast Ethernet. 1000Mbit/s Ethernet implementations are available as well, but switches remain expensive and thus have not yet experienced a high demand.

For parallel computation however, closely coupled systems are in use. Communication has to be very efficient in order to achieve reduced execution times. The shift from traditional network protocols to special own purpose protocols is still ongoing. Several applications can not benefit directly from

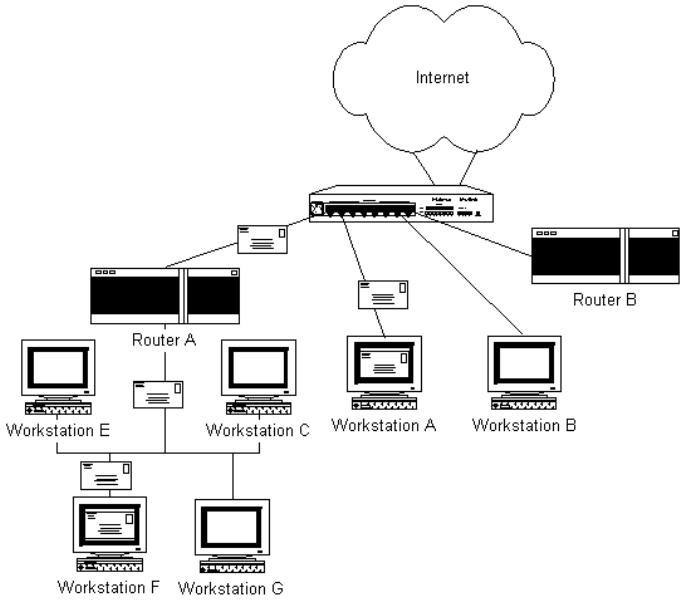

Figure 1.1: Overview on Local- and Wide Area Network

faster, proprietary networks, but need to be re-programmed or implemented using techniques such as data decomposition [120].

1.1.2 System Area Networks

System Area Networks (SANs) are a key component for boosting locally clustered distributed applications. Unlike standard, local area networks (LANs) based on Ethernet, SANs offer reliable communication with a Gigabit per second performance and latencies less than 10 micro seconds. While LANs are capable of bridging long distances between two communication endpoints at low speeds, the damping using high speed media is much higher and only allows for short distances. For SANs the maximum distance between two endpoints is about 100 - 200 meters when using fiber, LANs throughput can only be increased by parallel wires.

A SAN typically also offers multiple access ports. The network interface controller (NIC) exposes individual transport endpoints (ports) and demultiplexes incoming packets accordingly. Each endpoint is usually represented by a set of memory based queues and registers that are shared by

the host processor and the NIC. Many SAN NICs permit these endpoint resources to be mapped directly into the address space of a user mode process.

This reduces the *overhead* which has been determined to be one of the most costly factors involved with data exchanges [1], [2]. The communication takes place at the user level and only involves kernel traps for setting up, or releasing connections. This avoids unnecessary data copies from user address space to kernel address space for the delivery and reception of messages.

In order to handle requests directly from user mode applications, SAN NICs maintain page tables that map virtual addresses into physical addresses. Most of the current generation of interconnects require applications to register transfer buffers with the NICs driver which manages the page table. The driver pins buffers into physical memory while they are registered. A handle is associated with each registered memory region, and consumers must supply the handle corresponding to a data buffer in data transfer requests.

SANs offer two modes of data transfer. One is used mainly for small transfers and the other for large transfers. While the smaller ones are copied into registered buffers which are reused after transfers, bulk data transfer is performed through a Remote Direct Memory Access (RDMA) mechanism.

The initiator specifies a buffer on the local system and a buffer on the remote system. Data is then transferred directly between the two locations by the NICs without CPU involvement at either end. This mechanism typically requires a short rendezvous protocol in order to exchange address information.

In this context, System Area Networks are becoming more and more important for communication intensive applications but have not yet become an essential requirement, in part because existing applications can not directly benefit from the potential performance coming along with SAN.

This is because the current way of increasing application performance is to parallelize applications by using message passing libraries such as MPI or PVM. Furthermore, an immediate use of the SAN specific interfaces does not fit the data exchange model of existing applications. Thus, to gain performance, it would be required to re-implement already existing applications.

Providing an TCP/IP stack on top of a SAN is another solution. However, implementations of the TCP/IP stack on top of SANs [101] have shown that only 30 per cent of the raw performance can be achieved. This shows that the TCP/IP protocol, which was designed for unreliable connections in a WAN environment, is not suitable for achieving high performance in a cluster environment.

1.1.3 Communication Impact of Network Performance using LogP

The LogP Model [1] was developed as a realistic model for parallel algorithm design, in which critical performance issues could be addressed without being reliant on a variety of machine details. The performance of a system is characterized in terms of four parameters, three describing the time to perform an individual point to point message event and the last describing the number of processors involved, as follows.

- **L**atency - an upper bound on the time to transmit a message from its source to destination
- **o**verhead - the time period during which the processor is engaged in sending or receiving a message
- **g**ap - the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor
- **P**rocessors - the number of processors.

Another realistic assumption is the finite network capacity. This saturation can be reached if a processor is sending messages at a rate faster than the destination processor can receive. In the LogP model a processor which attempts to send a message that would exceed the finite capacity of the network stalls until the message can be sent without exceeding the finite capacity limit. These parameters are illustrated for a generic parallel system in Figure 1.2.

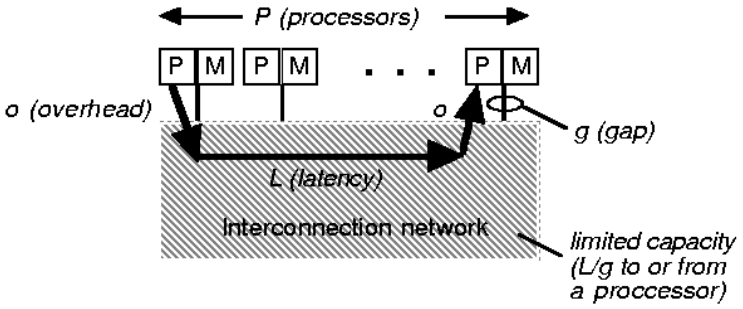

Figure 1.2: The LogP Model[2]

The total time for a message to get from the source processor to the destination is 2o + L. The *overhead* parameter reflects the time that the main processor is busy as part of the communication event, whereas the *latency* reflects the time during which the processor is able to do other useful work. The *gap* indicates the time that the slowest stage, the bottleneck, in the communication pipeline is occupied with the message. The reciprocal of the gap gives the effective bandwidth in messages per unit time.

Thus, transferring n small messages in a stream from one processor to another requires time o + (n-1) g + L + o, where each processor expends no cycles and the remaining time is available for other work. The same formula holds with many simultaneous transfers, as long as the destinations are distinct. However, if k processors send to the same destination, the effective bandwidth of each sender reduces to 1/(k*g). The overhead parameter is generally determined by the communication software and is strongly influenced by cost of accessing the NI over the memory or I/O bus on which it is attached. The latency is influenced by the time spent in the NI, the link bandwidth of the network, and the routing delays through the network. The gap can be affected by processor overhead, the time spent by the NI in handling a message, and the network link bandwidth. For a very large system, or for a network with poor scaling, the bottleneck can be the bisection bandwidth of the network. However, in practice the network interface is often the bottleneck for reasonably sized systems [86].

1.2 Clusters as a new Platform for Parallel Computing

Platforms for parallel computing can vary from massively parallel machines (Cray T3E, NEC SX), large shared memory processor system to networks of workstations. Recently cluster computing using standard components off the shelf (COTS) has become a viable platform for parallel computing. In this context, clustering means to harness the power of several individual nodes having the same hardware features. This approach is pursued since standard hardware components are available at a fraction of the cost but their aggregate performance can easily compete with massively parallel computers. As a comparison, the Heidelberg Linux Cluster (HELICS) consisting of 256 Dual AMD 1.4Ghz processors offers a sustained throughput of 825 GFLOPS measured with the High Performance Linpack (HPL) benchmark [134]. This performance has placed placed the cluster at the 35th position in the 19th Top500 list of the most powerful computers, which is compiled and published

twice a year [76]. The total cost of the cluster including a high speed network were approximately 1.2 Million Dollar, a comparatively small amount of money which has to be spent usually for achieving such high performance using proprietary hardware. For example, the 512 processor IBM p630 system at position 34 on the same list achieving 826 GFLOPS had the total cost of 6.4 Million Dollar, the Hewlett-Packard AlphaServer SC ES45 which was positioned at rank 37 achieving 809GFLOPS had a total cost of 24 Million Dollar (all systems were installed in 2002) [76], [133].

When building a cluster, single nodes can be equipped with one or more processing units. Typically a node consisting of two processing units show the best price performance ratio. In order to solve problems in parallel which requires communication among participating processes, a network is required for data exchanges. The type of network is the most crucial part of a cluster and therefore major aspects on how this network interfaces with a system at hardware and software level will be covered in section 1.3.

An operational cluster requires some kind of management software. Since single nodes of the cluster are physically independent, additional software will provide an abstracting layer of the underlying system. A resource management software will then be responsible for placing parallel applications on the nodes. A cluster can be typically partitioned and an application can request a subset of totally available nodes, leaving remaining nodes available for other applications.

These applications have been parallelized by a parallel programming model. A more detailed overview on available node architectures, as well as an introduction into available programming models will be presented in the following.

1.2.1 Shared- and Distributed Memory Systems

Individual nodes of a cluster can have a different physical appearance. A classification was given by Flynn [93], which identifies the Multiple Instruction Multiple Data (MIMD) class to be suited for cluster computing. The complete classification contains the following classes:

- SISD - Single Instruction Single Data
  a class of computers which consists of one processor. This processor has access to only one instruction and one data memory. During a computational step one instruction and its addressed data are fetched and processed. This architecture describes the von-Neumann computer, which corresponds to all sequential processing computers.

- MISD - Multiple Instruction Single Data
  a class of computers which has several processing units controlled by different instructions having only one common access to the data memory. In order to write data, either a result of one processor has to be chosen or the results have to combined. Due to that limitation no computer with such an architecture has ever been built.

- SIMD - Single Instruction Multiple Data
  a class of computers, in which all processing units are all controlled by the same instruction. But these instructions are applied to different data. Among others, all vector computers belong to this class.

- MIMD - Multiple Instruction Multiple Data
  a class of computers which includes all parallel computers having multiple processors. Each processor has its own private instruction and data memory. This leads to computers, which can process different data in different ways.

The following section will give an overview on how communication between several nodes can be achieved for machines belonging to the MIMD class.

When extending a traditional 1-CPU computer with further CPUs, a shared memory machine will offer the installed physical memory to any of these CPUs. This concept is also known as Unified Memory Access (UMA) , providing the same cost for each CPU when accessing data. A parallel application which is placed on this type of machines consuming several processors is able to communicate efficiently via IPC using chunks of data which have been exposed to be shared memory. While keeping a protected address space, one process can write to a buffer, while another one can read from it. For concurrent memory accesses, a synchronization mechanism like the MESI [104] protocol is required. In this situation the UMA architecture fails to scale well. Cost efficiency is reduced when larger numbers of CPUs per node are required. Another architecture type which implements shared memory is that of a of *Cache Coherent Non Uniform Memory Access* (CC-NUMA). It comes with varying latencies for memory accesses, including local memory (caches) for each processor. Depending on the application and its data structures, the access of nonlocal memory involves a higher cost.

A different concept is implemented as a distributed memory architecture in which every process has its own memory and the exchange of data has to be done explicitely via message passing. These data exchanges between processors require an interconnection network. Dependent on the features of

this network, a parallel application can experience more or less performance improvements.

1.2.2 Programming Models for Parallel Applications

Over the recent years, the development of programming models and their implementations has been very productive resulting in two major projects. With Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI), two portable reference implementations have brought the standardization of programming models. While other models such as OpenMP or Occam are using compiler directives, PVM and MPI provide mechanisms for explicit message passing, a model in which messages are sent (by the *source*) and being received (by the *destination*).

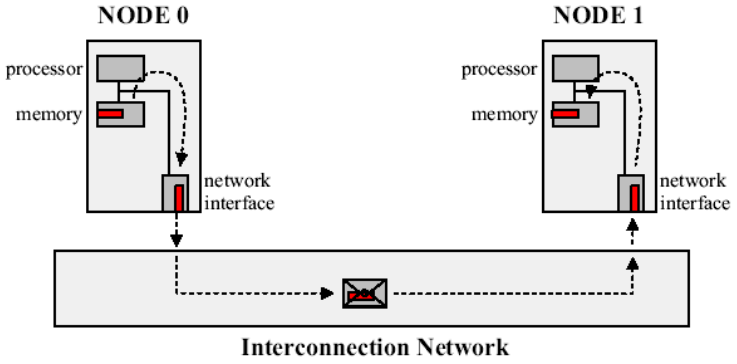

Figure 1.3: Message Passing in Distributed Memory Architectures. A typical scenario for a data exchange using two nodes of a cluster is depicted.

Figure 1.3 depicts an overview on how distributed Memory machines will exchange data using explicit messages using an interconnection network.

1.2.2.1 PVM

PVM (Parallel Virtual Machine) has been a popular message passing interface. It is an integrated set of software tools and libraries that emulate a general-purpose, flexible, heterogeneous, and concurrent computing framework on interconnected computers of varied architectures. The overall objective of the PVM system is to enable such a collection of computers to be

used cooperatively for concurrent or parallel computation. The PVM system is composed of two parts. The first part is a daemon, that resides on all the computers making up the virtual machine. When a user wishes to run a PVM application, first a virtual machine is created by starting up PVM in form of controlling daemons on each node. The PVM application can then be started from a Unix prompt on any of the hosts. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously. The second part of the system is a library of PVM interface routines. It contains a set of functions and primitives that are needed for cooperation between tasks of an application. This library contains user-callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine.

The PVM programming model is based on the notion that an application consists of several tasks. Each task is responsible for a part of the application's computational workload. Sometimes an application is parallelized along its functions; that is, each task performs a different function, for example, input, problem setup, solution, output, and display. This process is often called functional parallelism. A more common method of parallelizing an application is called data parallelism. In this method all the tasks are the same, but each one only knows and solves a small part of the data. This is also referred to as the SPMD (single-program multiple-data) model of computing. PVM supports either or a mixture of these methods. Depending on their functions, tasks may execute in parallel and may need to synchronize or exchange data, although this is not always the case. An exemplary diagram of the PVM computing model is shown in Figure 1.4.

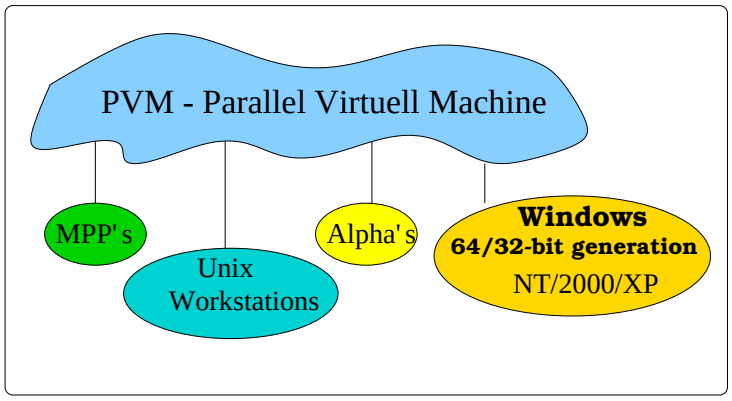

Figure 1.4: The PVM Model [119]

Briefly, the principles upon which PVM is based include the following:

- User-configured host pool: The application's computational tasks execute on a set of machines that are selected by the user for a given run of the PVM program. Both, single-CPU machines and hardware multiprocessors (including shared-memory and distributed-memory computers), may be part of the host pool. The host pool may be altered by adding and deleting machines during operation (an important feature for fault tolerance).

- Transparent access to hardware: Application programs either may view the hardware environment as an attributeless collection of virtual processing elements or may choose to exploit the capabilities of specific machines in the host pool by positioning certain computational tasks on the most appropriate computers.

- Process-based computation: The unit of parallelism in PVM is a task (often but not always a Unix process), an independent sequential thread of control that alternates between communication and computation. No process-to-processor mapping is implied or enforced by PVM; in particular, multiple tasks may execute on a single processor.

- Explicit message-passing model: Collections of computational tasks, each performing a part of an application's workload using data-, functional-, or hybrid decomposition, cooperate by explicitly sending and receiving messages to one another. Message size is limited only by the amount of available memory. Heterogeneity support: The PVM system supports heterogeneity in terms of machines, networks, and applications. With regard to message passing, PVM permits messages containing more than one datatype to be exchanged between machines having different data representations.

- Multiprocessor support: PVM uses the native message-passing facilities on multiprocessors to take advantage of the underlying hardware. Vendors often supply their own optimized PVM for their systems, which can still communicate with the public PVM version.

1.2.2.2 Message Passing Interface (MPI)

In 1993, a standard for a message passing environment was defined: MPI. MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any

implementation (such as a message buffering and message delivery progress requirement). MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes. Furthermore, MPI provides abstractions for processes at two levels. First, processes are named according to the rank of the group in which the communication is being performed. Second, virtual topologies allow for graph or Cartesian naming of processes that help relate the application semantics to the message passing semantics in a convenient, efficient way. Communicators, which house groups and communication context (scoping) information, provide an important measure of safety that is necessary and useful for building up library-oriented parallel code.

MPI also provides three additional classes of services: environmental inquiry, basic timing information for application performance measurement, and a profiling interface for external performance monitoring. MPI makes heterogeneous data conversion a transparent part of its services by requiring datatype specification for all communication operations. Both built-in and user-defined data types are provided.

MPI accomplishes its functionality with opaque objects, with well-defined constructors and destructors, giving MPI an object-based look and feel. Opaque objects include groups (the fundamental container for processes), communicators (which contain groups and are used as arguments to communication calls), and request objects for asynchronous operations. User-defined and predefined data types allow for heterogeneous communication and elegant description of gather/scatter semantics in send/receive operations as well as in collective operations.

MPI provides support for both the SPMD and MPMD modes of parallel programming. Furthermore, MPI can support inter-application computations through inter-communicator operations, which support communication between groups rather than within a single group. Dataflow-style computations also can be constructed from inter-communicators. MPI provides a thread-safe application programming interface (API), which will be useful in multi-threaded environments as implementations mature and support thread safety.

1.3 Network Interfaces

Over the last few decades, the location in which a network interfaces with the host system has experienced several changes. With the concentration on standard components, the design space on interfaces has been increasingly

restricted. This restriction led to the Peripheral Components Interconnect (PCI), which is a standardized bus interface used for most kind of I/O. In former times, the location of the network interface was much more flexible. Since it is the CPU which is triggering communication, a network device is more efficient the closer it can be attached to the CPU.

Amdahls law, which observes that the processing performance doubles every 18 months, is still valid in the 21st century. However, the performance of the I/O bus architecture has only doubled every 36 months. Hence there is an increasing gap which is illustrated over the recent years in Figure 1.5.

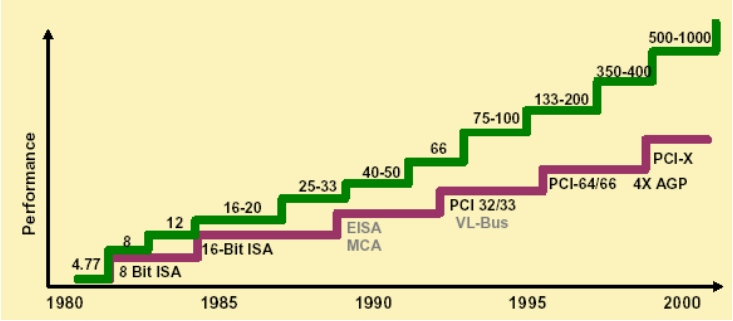

Figure 1.5: Overview on Microprocessor and I/O Bus Performance[90]

This bottleneck is slowing down system performance and high speed networking technologies. System Area Networks and emerging standards like Infiniband will not experience the performance they need in order to achieve their performance.

The design space of network interfaces can still be open to new approaches although the continuous hardware standardization efforts limit actual implementations. A detailed overview on the design spaces of network interfaces can be found in [84]. It also serves as a reference to implementations which are no longer available. Current interfaces do not implement I/O sharing, provide kernel bypass support, memory protection and other higher level functions to external fabrics. The following provides an introduction to current and future interfaces and the impact of this road map on further communication primitives which are the subject for innovative communication mechanisms.

1.3.1 PCI 2.x and PCI-X as Current Interfaces

The PCI bus was developed in the early 1990s by a group of companies with the goal to advance the interface allowing OEMs or users to upgrade the

I/O (Input-Output) of personal computers. It essentially defines a low level interface between a host CPU and peripheral devices. The PCI architecture utilizes PCI to PCI (P2P) bridges to extend the number of devices that can be supported on the bus. By definition a system built from P2P bridges forms a hierarchical tree with a primary bus extending to multiple secondary PCI bus segments. Across all of these PCI bus segments there is a single physical memory map. A given memory address uniquely specifies an individual bus segment and device on this segment. This architecture fundamentally defines a single global view of resources, which works well to build systems based on a single master (host CPU) sitting at the top of the hierarchy controlling multiple slaves on peripheral bus segments. In this case master and slave refer to the initialization and control of the devices in the system rather than to the capability of an individual slave device to initiate a bus transaction (i.e. acting as a PCI bus master). The PCI bus has proven a huge success and has been adopted in almost every PC and Server since. The latest advancement of the PCI bus is PCI-X. PCI-X is a 64-bit parallel interface that runs at 133 MHz enabling 1GBytes/s (8Gbits/s) of bandwidth. Though other advancements are under development, including DDR, for the PCI bus, they are perceived as falling short. They are too expensive (too many pins in the 64-bit versions) for the PC industry to implement in mass volume they fail to offer sufficient bandwidth and advanced feature set required for the servers of the future. Further information and a detailed description on the benefits of PCI-X compared to PCI can be found in [85].

With PCI 2.2, increased interface width and frequency can already yield to improved performance. A 64-bit PCI bus provides higher overall throughput for high-performance adapters and better system efficiency by providing the same data in fewer PCI clock cycles. A 66-MHz PCI bus doubles the data throughput over the same amount of time. The benefits of both 64-bit and 66-MHz PCI implementations are better PCI bus utilization, better overall PCI bus efficiency, and a substantial increase in PCI bus performance. In our tests we have however measured a quite large discrepancy among several boards. It is evident that the implementation of a chipset for a 64bit interface is not a problem, giving an average sustained performance of 220MBytes/s for a 33Mhz device (approximately 85% efficiency), but it is the 66Mhz version which achieves good performance only on advanced chipsets. The larger fraction reaches approximately 60% efficiency when running in 66Mhz mode. Another specification which is currently being discussed is that of PCI-X 266/529. This is touted as a straight forward technology. The specification uses the same 133MHz clock but simply clocks the data on both rising and falling edge of the clock to double the effective bandwidth to 266 MHz. PCI-

X 266 uses much the same technology as memory and system vendors used to implement DDR SDRAM. Proponents estimate that this technology can be implemented in 2002, while others raise questions about how robust the backwards compatibility will be.

Still, the implications of the PCI model when designing a network interface card are tremendous. PCI does not allow for very efficient communication performed solely by hardware and therefore it is the software layer which can serve as a point of investigation.

1.3.2 Future Interfaces

With the dramatic changes over the recent years and an ongoing trend towards components off the shelf (COTS), it is most likely that future interfaces will keep current limitations.

1.3.2.1 PCI-Express

In August 2001 at the Intel Developers Forum it was announced that PCI-Express (formerly known as 3GIO) would be developed as a new local bus (chip-to-chip interface) and used as a way to *upgrade* the PCI bus. The PCI-Express architecture is being defined by the Arapahoe Working Group and upon completion, it will be turned over to the general PCISIG organization for administration. PCI-Express is defined as serial I/O point-to-point interconnect. Basic layer consists of a dual-simplex channel that is implemented as a transmit pair and a receiver pair. The intent of this serial interconnect is to establish very high bandwidth communication over a few pins, versus low bandwidth communication over many pins (as in the 64-bit PCI interface). It also leverages the PCI programming model to preserve customer investments and to facilitate industry migration. This allows PCI bandwidth to be economically upgraded without consuming a great number of pins while preserving software backwards compatibility. The stated goal of PCI-Express is to provide:

- A local bus for chip-to-chip interconnects
- A method to upgrade PCI slot performance at lower costs

1.3.2.2 HyperTransport

The HyperTransport technology is intended to provide a high-speed, high performance, point-to-point link for interconnecting integrated circuits on a board.

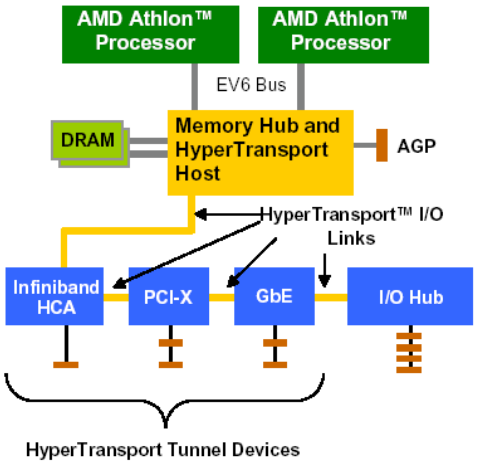

Figure 1.6: HyperTransport and Tunneling Devices [90]

With a top signaling rate of 1.6 GHz on each wire pair, a HyperTransport technology link can support a peak aggregate bandwidth of 12.8 GBytes/s. The HyperTransport I/O link is a complementary technology for InfiniBand and 1Gb/10Gb Ethernet solutions. While InfiniBand and high-speed Ethernet interfaces are high-performance networking protocol and box-to-box solutions, HyperTransport is intended to support in-the-box connectivity. This however may change in the near future. On June 20th, 2002, the HyperTransport Consortium [90] released a document on network extensions to HyperTransport to allow for box to box communication.

Figures 1.6 and 1.7 depict a host system which has tunneling devices to allow for example for interconnecting other networks, but also shows HyperTransport usage for building cache coherent systems.

Still, the HyperTransport specification provides both link- and system-level power management capabilities optimized for processors and other system devices. The Advanced Configuration and Power Interface (ACPI) compliant power management scheme is primarily message-based, reducing pin-count requirements. HyperTransport technology is targeted at networking,

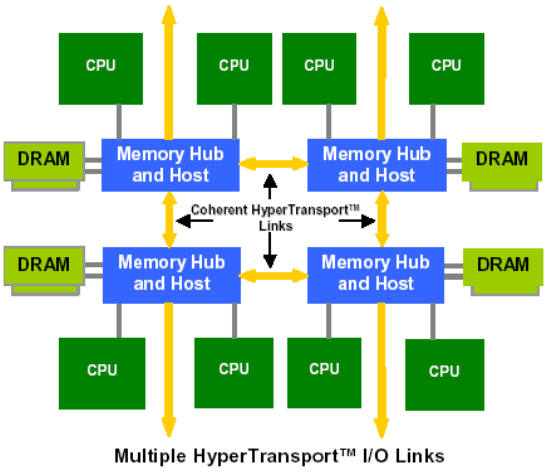

Figure 1.7: HyperTransport in Cache Coherent SMP Systems [90]

computer and high performance embedded applications and any other application in which high speed, low latency, and scalability is necessary. For further and more detailed information which are beyond this introduction, the white papers and the specifications at [90] are a good reference.

1.3.2.3 Infiniband

InfiniBand [33] is fundamentally different to PCI, as devices are designed to operate as peers with channels, which are also named queue pairs (QP)s. These channels may each have their own independent Virtual and Physical Address spaces. This allows any node to be an initiator to any other node throughout the fabric. The InfiniBand architecture provides a large QP space to support up to 16 million channels, and every channel is capable of delivering a reliable, low latency, transport level connection. Such a channel based architecture functions ideally as a multi-protocol I/O fabric by providing fine grained quality of service to each channel which allows application level optimizations.

1.4 Goal of this Thesis

The solution of large computational problems requires its decomposition onto multiple resources to achieve an answer in a reasonable amount of time. Typ-

ically this decomposition requires communication among the processors involved. The result of this parallelization effort ranges from embarrassingly parallel jobs to fine granular applications which require frequent communication. The latter serves as a motivation for this thesis. Frequent communication is required for a large set of applications and the communication capabilities have the most impact on lowering the execution time. The goal of this thesis is to improve traditional concepts on networking and to develop new, innovative protocols which will overcome existing bottlenecks. This work also includes the optimization of mechanisms which can be found along the communication path.

For this, the optimization can be done at two different levels, the software and hardware layers. For the software layers, a plugin for the Parallel Virtual Machine (PVM) will enable the usage of fast networks with no additional protocol overhead. This will lead to high performance communication on clustered systems. The plugin will also maintain its usage for heterogeneous environments in which traditional protocols will be used. Another feature of this plugin will be its transparency. No modifications are required in order to experience much faster communication throughput using high speed networks.

This thesis however also extends the usage of System Area Network (SAN) to a completely new area. Currently SANs are considered being a special type of network, requiring special handling. With the introduction of a new transparent messaging layer, this thesis will map any distributed application to a SAN transparently. It will even achieve *binary compatibility*. The most important features of this innovative layer are that the time for developing an existing application is saved since the application does not have to be modified or redesigned and implemented again to use a new programming library for communication. Moreover, an existing application will experience an order of magnitude performance improvement. As an example, a set of benchmarks will be presented that will show a performance improvement (peak values) from 80MBytes/s to 200MBytes/s. This layer will also lower communication overhead. One way latency has been reduced from more than 100 usec to 13 usec. For a real world application the performance improvements have been demonstrated on a commercial data base. The installation of this proprietary data base does not offer any source code and binary compatibility for message layers is an absolute requirement. As a result, a 35% improvement for the number of transactions for a given amount of time was achieved by mapping the application to the developed layer.

These software layers require an interconnection network. Based on the availability of existing and future high speed network this thesis will make

clear, that communication protocols are the key for achieving reasonable throughput. Based on the ATOLL network card hardware [83], which was introduced in [82], the message passing protocols MPI and PVM were implemented. Based on the given hardware, the ATOLL network was then extended to provide a network interface which is capable of efficient communication. It is another subject of this thesis to show the need for direct data transfers, in which data copies are avoided and the host system is left out of the transfer of bulk messages. For this a comparison on different transfer models will be given and compared with developed RDMA supported data transfers. RDMA capability has been developed for the extended ATOLL network interface.

1.5 Thesis Organization

This thesis is organized as followed. Chapter 2 will analyze the impact of communication on parallel computing. It will differentiate between traditional communication primitives involving the host system and new protocols which bypass the operating system enabling user level communication. It will also analyze which design space for network interfaces is available in general. A hardware overview on traditional networks and system area networks follows. Finally a performance analysis for applications using different hardware and protocols concludes this chapter. In Chapter 3 a system area plugin for the Parallel Virtual Machine will be presented. This new work allows PVM for fast communication while keeping all PVM primitives such as dynamic process management for starting and ending processes and heterogeneity. It will specify a very basic interface in which a new low level API of a SAN can be developed. The plugin has been provided for todays popular SANs. Furthermore, a general description on how SANs can be used efficiently will be presented. A performance analysis of the different plugin's concludes this chapter. Chapter 4 will describe the developed communication environments for the Atomic Low Latency (ATOLL) network. Traditional message passing environments have been ported to ATOLL. The further development of a ATOLL API was performed concurrently to allow for an efficient integration into message passing environments. In Chapter 5 the design for an advanced ATOLL network will be presented. With this design, a flexible protocol will be introduced which still can be realized as protocols in hardware. It will first give a motivation by showing the benefits for an application and a host system using RDMA capable network hardware. It will then describe how protocols can be integrated into the existing ATOLL framework. This design

is evaluated using different strategies. It will determine that an extension to ATOLL offers major enhancements in comparison with a software approach.

Finally, Chapter 6 will present a new middleware layer which is capable of replacing traditional network protocols with compatibility at binary level. It will provide an analysis of these protocols and explain why a new approach is required. It will then describe which techniques given todays operating system features are available in order to achieve this replacement. This requires a full understanding about the functionality of applications and operating systems. This new middleware layer is able to let application fully exploit the available performance. It will offer a performance increase in the order of a magnitude. This will be verified by several benchmarks. Moreover, this middleware layer is also beneficial for real world applications. It will open new uses for system area networks. For this the improvement of a transaction database will be presented. The test environment consists of application servers querying a database server. In this scenario, the new middleware layer will demonstrate its efficiency by increasing the transaction throughput by 35%.

Chapter 2

Impact of Communication on Distributed Computing

This chapter will describe how the combination of software layers and the underlying hardware have an influence on distributed computing. The chapter will consist of a separated software and hardware section. The software section will deal with communication protocols and their overhead, while the hardware section will describe existing networks on which different software stacks will be based.

2.1 Software

Communication requires software stacks which will be talking to the networking device. A major shift in the communication paradigm has been seen in the recent years. This section will give an overview on current state of the art techniques and their implications. Basically, software stacks, under the control of the de-multiplexing operating system, are required for traditional networking concepts.

2.1.1 System and User Level Modes

A computer is nowadays under the control of an operating system (OS) [132]. The OS is based on a set of programs, which will maintain the functionality of a computer, independent of the actual application. It furthermore controls hardware resources and provides a hardware abstraction layer (HAL) in order to hide the complexity of the underlying hardware.

The tasks of an OS are:

- Process Scheduling: fair scheduling of multiple processes, which are running on a computer
- Memory Handling: Allocation and Protection of Main memory among different processes.
- Hardware management: Avoiding access collisions of different process to the same hardware as well as file management.

In order to achieve the mentioned tasks, the operating system introduces different levels. Processes can run in user level or kernel level mode offering varying levels of protection. Applications typically run as user level processes and will call a system function which results in a kernel trap. The kernel will then perform requested tasks of the application such as accessing the network device for example. This is required since on traditional systems, the network device can be accessed by multiple processes. Since a multitasking system allows for several processes to be run at the same time, several of them can have an open port to the networking device, but only one is granted access to avoid data corruption but also to maintain security.

Other advantages of concept are for example the improved stability. Otherwise scenarios could be possible in which an application which is accessing a device experiences a segmentation fault and can not release the device any more. This way, no other device would be able to get access to the device again. The disadvantages are that system traps are a costly operation and a process can be put into wait state when accessing a device which is currently in use. For this multiple devices would improve this situation so that several applications could for example communicate at the same time.

This leads to the concept of user-level networking in which several devices are available and an application can get access to its reserved device for direct communication. User-level networking schemes expose a lower-level abstraction of the network device than standard protocols. Applications are given their own virtual interface that they can manipulate to queue packets for transmission and reception. Bypassing conventional operating system mechanisms in this way allows applications to reduce the latency and per-packet overhead of using the network. The motivation for much of this work has been parallel processing on workstation clusters, where minimal round-trip latency is critical. Where user-level network access is to be offered to applications in a multi-user, multi-programmed environment, care must be taken to avoid sacrificing the sharing and system protection previously performed by the operating system. Along with the protection of memory that applications use for network transfers, it is also necessary to prevent applications from

snooping messages distinct for other programs, and from spoofing messages such that they appear to originate from other sources. Existing user-level networking efforts have addressed connection-oriented networks such as ATM [88], [61] and special-purpose System Area Networks (SANs) [56], [9]. Establishing a virtual circuit between two end-points provides a convenient means of securely identifying the source and destination applications of transmitted data. This connection establishment as it is required for example for the Virtual Interface Architecture (VIA) network or the ATOLL network, comes with a sequence of `connect` functions which have to be called by the two endpoints. Such a `connect` sequence can then establish routing information, or provide endpoint specific information, if accepting any endpoint partner.

User-level interfaces then need only ensure that applications are restricted to sending and receiving on virtual circuits that they own. Connectionless datagram networks such as Ethernet pose a greater problem. Where user-level interfaces have previously been developed for connectionless networks such as Fast Ethernet, they have relied on modified versions of standard protocols to allow simple de-multiplexing at the receiver, and are thus unable to inter-operate with other systems.

2.1.2 Implications of User Level Communication

If applications obtain direct access to hardware, the notion of opening a *port* is used. In this context, a port is the direct access point to the underlying hardware. Typically several ports exist per item and the hardware itself can deal with assigning more than one direct access point. While direct access is granted for processes to lower the cost for data exchanges, it also requires thorough handling of operations on the device. Unmeant interference with other processes must be avoided.

Basically, the concept of direct hardware access goes back to basic operating systems like DOS. Even in recent Windows (TM) versions like Windows 95, 98 and Me (TM) processes can access the hardware directly. Only their professional versions introduce a hardware abstraction layer, the process is required to talk to. The following section discusses the aspects of security, functionality and stability.

2.1.2.1 Security

First concepts and early user level communication protocols have not provided security for other processes. This means that direct hardware accesses

did interfere with other with other user processes. This can easily happen when for example an application experiences a stack overflow and would write on user level data which describes descriptors to be accessed by the hardware. For current standard network devices, autonomous DMA engines operate on physical addresses and no protection can be given by the OS anymore. This however could lead to unpredictable behavior This however is now secured by the low level API of a System Area Network together with additional checks of data regions by the hardware itself.

2.1.2.2 Functionality

Traditional protocols are communicating by calling system functions. They involve a trap into the operating system and the application is blocked until the system call has finished. Not only the system trap is rather costly, but also the actual request may not be scheduled immediately. For example, a send request will first inserted into operating system data structures and scheduled for delivery. When direct access to hardware is given, the application will use data structures which are in sync with the network device to interoperate with the device. This way, the overhead will be reduced by avoiding data copies, but the application will also experience lower latency.

2.1.2.3 Stability

State of the art of user level communication protocols have added additional efforts to allow for a stable operation of direct access networks. This approach has resulted in incorporating important tasks into the operating system. For different Unices like Unix or Linux, low level drivers have been implemented as *modules*. For these OS, a module becomes part of the OS by attaching itself to a operating system dynamically during runtime. As a module, low level drivers become part of the operating system and could bring a system down when for example operating incorrectly on internal virtual memory structures. Modules must be loaded with *root* permissions and must itself provide several functions to be manageable by the OS. These restrictions as well as the concept of a single instance has easened the issue of portability as well since modern OS like Linux tend to change dramatically over a short period of time. Moreover, most of the low level SAN drivers are distributed as Open Source, so that the community can have a look at them or submit improvements. The SCI network [109] did not gain success because of their proprietary driver concept. Early driver were buggy and the uptime of a node was less than a day. One reason for this was the immediate interface

between the SCI network and the host memory system which led to constant crashes. An open source implementation would have achieved shorter time for bug fixes.

2.1.3 User Level Communication Protocols

Several user level communication protocols have been introduced over the recent years. The following sections describe existing implementations to better understand the concept of user level networking. In upcoming sections, low level protocols will be used for replacing traditional network protocols. Therefore a thorough understanding is required.

2.1.3.1 Active Messages

The Active Message (AM) communication layer provides a collection of simple and versatile communication primitives. It is generally used in libraries and compilers as a means of constructing higher-level communication operations such as MPI [102], [103]. AM can be seen as a very lightweight asynchronous remote procedure call, where each operation is a request/reply pair. In LogP terms, an AM request/reply operation includes two point-to-point messages, giving an end-to-end round-trip time of $2(o_s+\mathrm{L}+o_r)$. A request message includes the address of a handler function at the destination node and a fixed number of data words, which are passed as arguments to the handler. AMs are handled automatically, either as part of the node initiating its own communication, via an interrupt, or as part of waiting for responses. Otherwise, a node can also handle messages via an explicit poll. When the message is received at the destination node it invokes the specified handler, which can perform a small amount of computation and issue a reply, which consists of an analogous reply handler function and its arguments. This basic operation is illustrated in Figure 2.1 by typical remote read transaction.

AM is efficient to implement because messages can be issued directly into the network from the sender and, since the code that consumes the data is explicitly identified in the message, processed directly out of the network without additional buffering and parsing. The handler executes in the context of a prearranged remote process and a fixed set of primitive data types are supported, so the argument marshaling and context switching of a traditional RPC are not required. The sender continues execution as soon as the message is issued and the invocation of the reply handler provides notification of completion.

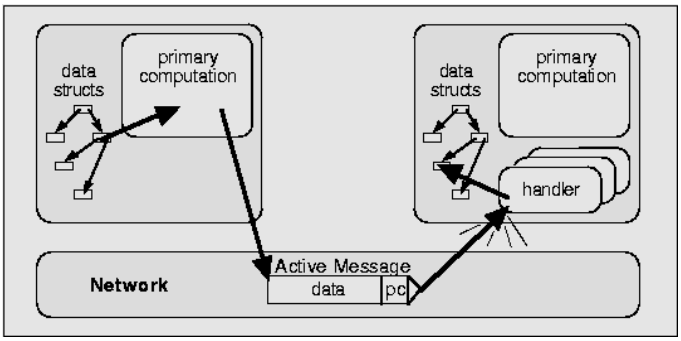

Figure 2.1: Active Message Model [105]

The Active Messages II performance measured on the NOW cluster [94] was very good [115]. It achieved 43.9 MBytes/s bandwidth for 8 KB messages, that is about 93% of the 46.8 MBytes/s hardware limit for 8 KB DMA transfers on the Sbus. The one-way latency for short messages, defined as the time spent between posting the send operation and message delivery to destination endpoint, is about 15 micro seconds.

Similar to Active Messages, Remote Queues [12] provided low-overhead communications, but separated the arrival of messages from the invocation of handlers.

2.1.3.2 VIA

In this paragraph, an overview of the VI Architecture is presented. VIA derived from a large body of related work in user-level communication. VIA borrows its basic operation from U-Net [26]. Virtual interfaces to the network were introduced by application device channels [22], and remote memory operations were taken from the Virtual Memory Mapped Communication (VMMC) [62] model and from Active Messages (AM) [60].

2.1.3.2.1 VIA Overview Figure 2.2 depicts the organization of the Virtual Interface Architecture. In the following, essential information from the VIA specification is reflected. For further information, it can be downloaded from [56]. Basically, the VI Architecture is comprised of four basic components: Virtual Interfaces (VI), Completion Queues, VI Providers, and VI Consumers. In VIA, the VI Provider is composed of the VI Network Adapter and a Kernel Agent device driver. The VI Consumer is composed of an application program and a high level communication subsystem such as PVM,

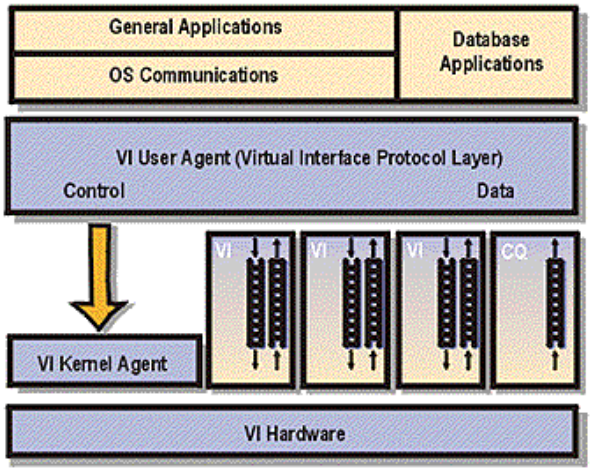

Figure 2.2: VIA Architectural Overview

MPI or sockets. Other applications communicate with the VI Provider API directly (e.g: Oracle 9i or IBM DB2 Databases). Prior to communication, a connection setup by the Kernel Agent is required. Using a pair of virtual channels, all network actions occur without kernel intervention. This results in significantly lower latencies than network protocols such as TCP/IP. Traps into kernel mode are only required for creation/destruction of VIs, VI connection setup and teardown, interrupt processing, registration of system memory used by the VI NIC, and error handling. VI Consumers access the Kernel Agent using standard operating system mechanisms.

For each VI one Send- and Receive Queue is created. VI Consumers post requests for sending or receiving data in form of descriptors to these queues. Such a descriptor holds all required information that the VI Provider needs to process the request, including source, destination, message length and pointers to data buffers. VI Providers asynchronously process the posted descriptors and mark them when completed. VI Consumers remove completed descriptors from the Send and Receive Queues and reuse them for subsequent requests. Both the Send and Receive Queues have an associated doorbell that is used to notify the VI network adapter that a new descriptor has been posted to either the Send or Receive Queue. The implementation of doorbells is not fixed. For efficiency, the doorbell can be directly implemented on the VI Network Adapter and no kernel intervention is required to perform this signaling. The Completion Queue allows the VI Consumer to combine the notification of descriptor completions of multiple VIs with-

out requiring an interrupt or kernel call. In order to eliminate the copying between kernel and user buffers that accounts for a large portion of the side, overhead associated with traditional network protocol stacks, the VI Architecture requires the VI Consumer to register all send and receive memory buffers with the VI Provider. This registration process locks down the appropriate pages in memory, which allows for direct DMA operations into user memory by the VI hardware, without the possibility of an intervening page fault. After locking the buffer memory pages in physical memory, the virtual to physical mapping and an opaque handle for each memory region registered are provided to the VI Adapter. Memory registration allows the VI Consumer to reuse registered memory buffers, thereby avoiding duplication of locking and translation operations. Memory registration also takes page-locking overhead out of the performance-critical data transfer path.

2.1.3.2.2 Data Transfer Modes The VI Architecture provides two different modes of data transfer: traditional send and receive semantics, and direct reads and writes to and from the memory of remote machines. Remote data reads and writes provide a mechanism for a process to send data to another node or retrieve data from another node, without any action on the part of the remote node (other than VI connection). The send/receive model of the VI Architecture follows the common approach to transferring data between two endpoints, except that all send and receive operations complete asynchronously. The VI Consumers on both the sending and receiving nodes specify the location of the data. On the sending side, the sending process specifies the memory regions that contain the data to be sent. On the receiving side, the receiving process specifies the memory regions where the data will be placed. The VI Consumer at the receiving end must post a Descriptor to the Receive Queue of a VI before the data is sent. The VI Consumer at the sending end can then post the message to the corresponding VI s Send Queue. Remote DMA transfers occur using the same descriptors used in send/receive style communication, with the memory handle and virtual address of the remote memory specified in a second data segment of the descriptor. VIA-compliant implementations are required to support remote write, but remote read capability is an optional feature of the VIA Specification.

2.1.3.2.3 VIA Implementations VIA is available as hardware and software implementation. Hardware implementations are for example the GigaNet cLAN GNN1000 network interface card (available through Emulex

with a focus on storage), and Servernet II from Compaq, Software implementations are available through M-VIA which is abstracting from specific hardware and even runs on Ethernet cards. More efficient implementations are also available on top of low level drivers for todays System Area Networks. It may be also worth noting that the VIA will be the standard interface used for the upcoming Infiniband technology.

2.1.3.3 U-Net

U-Net is a research project which started in 1994 at Cornell University. The goal was to define and implement a user-level communication system for commodity clusters of workstations. The first experiment was done on 8 SunOS SPARC-Stations interconnected by the Fore Systems SBA-200 ATM network [61]. Later, the U-Net architecture was implemented on a 133 MHz Pentium cluster running Linux using Fast Ethernet DC21140 network interfaces [113]. The U-Net architecture virtualizes the network interface, so that every application can think of having its own network device. Before a process can access the network, it must create one or more endpoints. An endpoint is composed of a buffer area which holds message data and message queues for sending, receiving and freeing. These queues contain descriptors for messages that are to be sent or have been received. The buffer area is pinned down and virtual addresses are translated into physical addresses to be used by DMA engines. Message descriptors contain source, destination, message length and offsets within the buffer area for referring to specific message data. The free queue contains descriptors to hold pointers to free buffers to be used for incoming data. Two endpoints communicate through a communication channel, distinguished by an identifier that the operating system assigns at channel creation time. Communication channel identifiers are then used to generate tags for message matching. To send a message, a process puts data in one or more buffers of the buffer area and inserts the related descriptor in the send queue. Very small messages can be insert directly in descriptors. The U-Net layer adds the tag identifying the sending endpoint to the outgoing message. On the receiving side U-Net uses the incoming message tag to determinate the destination endpoint, moves message data in one or more free buffers pointed by descriptors of the free queue and put a descriptor in the process receive queue. Such descriptor contains the pointers to the just filled buffers. The destination process is allowed to periodically check the receive queue status, to block until the next message has arrived, or to register a signal handler with U-Net to be invoked when the receive queue becomes non-empty. U-Net has been implemented on different platforms and

some of the measured performance will be presented in the following. The U-Net/ATM performance is very close to that of the raw SBA-200 hardware (155 Mbit/s). It achieves about 32 micro seconds one-way latency on short messages and 15 MBytes/s asymptotic bandwidth.

2.1.3.4 Fast Messages

Fast Messages (FM) is a communication system developed at University of Illinois. It is very similar to Active Messages and was originally implemented on distributed memory parallel machines, in particular the Cray T3D. Later, it was ported to a cluster of SPARC-Stations interconnected by the Myrinet network [114]. In both cases the design goal was to deliver a large fraction of the raw network hardware performance to user applications, paying particular attention to small messages because these are very common in communication patterns of several parallel applications [118]. Fast Messages is targeted to compiler and communication library developers, but application programmers can also use it directly. FM provides few basic services and a simple programming interface. The programming interface consists of three functions for sending short messages (4-word payload), for sending larger messages than 4 words and for receiving messages. Like Active Messages, each message contains a pointer to a sender-specified handler function that consumes data on the receiving processor. No request-reply mechanism is provided. It is the programmers responsibility to prevent deadlock situations. Fast Messages provide buffering mechanisms allowing sending processes to continue computation while their corresponding receivers are not servicing the network. The receiver calls the FM_extract() function to retain data from a buffered message queue. The function checks for new messages and executes provided handlers. It is a requirement for the receiver to periodically call the FM_extract() function to ensure the prompt processing of incoming data. However, not doing so will not prevent the network from making progress. Furthermore, the Fast Messages design assumes that the network interface has an on board processor with its own local memory, so that the communication workload can be divided between host and network processor. Such an assumption allows Fast Messages to efficiently expose two main services to higher level communication layers, control over scheduling of communication work and reliable in-order message delivery, respectively. Reliable in-order message delivery prevents the cost of source buffering, timeout, retry and reordering in higher level communication layers, requiring Fast Messages only to resolve issues of flow control and buffer management, because of the high reliability and deterministic routing of the

Myrinet network. Fast Messages 2.x was implemented on the High Performance Virtual Machine (HPVM) cluster. It consisted of 256-nodes running the Windows NT operating system, interconnected by Myrinet [9]. Each node had two 450 MHz Pentium II processors. A 8.8 micro seconds one-way latency for zero-payload packets and more than 100 MBytes/s asymptotic bandwidth were measured.

2.2 Design Space for Network Interfaces

In this section we would like to evaluate current network interfaces and characterize the design space for I/O subsystems in general. A first distinction can be made by dividing I/O subsystem components into hardware and software components.

A special purpose processor, additional (staging) memory, support for overlapping data transfers, PIO and DMA operations and support for shared memory reflect important hardware features. Partly they are available onboard of the network interface card.

In addition, switching hardware is needed to build up large scaling clusters.

Data transfer protocols and their implementations are then a central part of the software component. Support for Remote Direct Memory Access (RDMA), direct memory management unit (MMU) support, the efficient detection of message delivery and arrival are key factors for gathering high performance.

We would like to break down the design space into the following items:

- Concurrency through PIO and DMA Transactions
- MMU Functionality to support RDMA

When sending a message, the low level Application Programming Interface (API) chooses PIO or DMA modes for data transfer. The preferred mode is depending on the message size. PIO has the advantage of low start-up costs to initiate the transfer. However, since the processor is transferring data directly into the network, it is busy during the entire transaction. To allow concurrency, the DMA mode must be chosen in which the processor only assembles a descriptor pointing to the actual message. This descriptor is then handed to the DMA engine which picks up the information and injects the message into the network. It is important to know that the DMA engine relies

on pinned down memory since otherwise pages can be swapped out of memory and the engine can not page on demand by itself. The advantage of using DMA is to overlap communication and computation. However it has a higher start-up time than PIO and thus, usually a threshold values determines which protocol is chosen. Both mechanisms also play an important role when trying to avoid memory copies.

2.2.1 Intelligent Network Adapter, Hardware and Software Protocols

The most important feature by having an intelligent network adapter (processor and SRAM on board) is to be flexible in programming the message handling functionality. Protocols for error detection and correction when exchanging a message can be programmed in software, but also new techniques can be applied [56]. Support for concurrency is improved as well. Additional memory on board lowers congestion and the possibility of deadlocks on the network. It has the advantage to buffer incoming data, thus emptying the links on which the message has been transferred. However, the memory size is usually limited and expensive and the number of data copies increases. Another disadvantage of this combination is that the speed of an onboard processor resource can not cope with the main processing unit. Moreover, the programming of custom network adapter can be a versatile task since standard software environments are missing.

2.2.2 Switches, Scalability and Routing

A point-to-point micro benchmark typically only shows the best performance for non-standard situations. Since a parallel application usually consists of dozens of processes communicating in a more or less fixed pattern, measuring the bisection bandwidth generates better information of the underlying communication hardware. A cost-effective SAN typically consist of one or more bidirectional links that allow for concurrent send and receive data transfers. These links use standard IO cells and throughput between two nodes can be increased by bonding several links. This technique however requires additional support to be implemented in the low level API. If data travels over different links, sequence numbers have to be assigned which describe in which order data has to be received. A key factor for gaining superior performance is the scalability. In this situation switches are added to build a multistage connection network to allow for larger clusters. Another point of interest

is the connection from NIC to NIC: Data link cables must provide a good compromise between data path width and transfer speed.

2.2.3 Hardware support for Shared Memory (Coherency) and NI locations

Currently a trend can be seen in clustering bigger Shared Memory Processors (SMPs) nodes with 2, 4 up to 16 processors per node. Within an SMP node a cache coherency protocol like MESI (Modified, Exclusive, Shared, Invalid), where each cache line is marked with one of the four states, will determine when data needs to be synchronized.

Obviously, a cache coherent NIC requires to participate on the cache coherent protocol, which is only possible by snooping on the system bus. However, this would involve a special solution and therefore can not be propagated as a commodity solution. With the growing distance between the NI and the processor the latency of the communication operations raises and, at the same time, the bandwidth declines. The only position that results in a wide distribution and, thus, the necessary high quantities for the NIC, is the standardized PCI bus. This leads to the loss of a number of functions, like the cache coherent accesses to the main memory of the processor. As the NI on the PCI card is independent from the used processor (and has to be), functions like the MMU in the NI cannot be easily synchronized, as they differ according to which processor is being used. In the NIC the necessary functions must be realized as efficiently as possible to keep the overhead of the software low. For this purpose a direct hardware realization of the basic mechanisms, as in the ATOLL-SAN, or an additional processor on the PCI card, as implemented in the Myrinet network, can be used.

2.2.4 Performance Issues: Copy Routines and Notification Mechanisms

Once a message is ready for sending, the data has to be placed at a location where the NI can fetch the data. Using standard copy routines however can result in poor performance. The reason is that the cache of the CPU spilled with data copies when larger messages have been injected into the network. Modern CPUs like the Pentium III/IV or Ultrasparc offer special multi-media extensions (MMX) or Visual Instruction Set (VIS) instructions, which copy the data without traversing the cache. Another point which can hamper performance is how the arrival of messages is detected. A polling

method typically wastes a lot of CPU cycles while an interrupt causes too much overhead. The latter introduces its overhead by high context switching. Avoiding the interrupt mechanism is very important as each new interrupt handling leads to a latency of approximately 60 micro seconds [121].

2.3 Hardware

In order to achieve high communication performance results, software stacks must be implemented as thin layers which hand data to the underlying hardware without introducing additional protocol overhead. By adapting user level communication to a variety of network interfaces, this effort has become a standard way of implementing efficient communication layers. The overall performance however, is also very dependent on the underlying hardware with respect to its physical capabilities and limitations. In order to develop new middleware protocols, it is required to have an understanding of available network interface implementations and how they differentiate themselves. The most popular implementations will be presented in the following.

2.3.1 Fast- and Gigabit Ethernet

2.3.1.1 Overview

Gigabit Ethernet, also known as the IEEE Standard 802.3z, is the latest Ethernet technology. Like Ethernet, Gigabit Ethernet is a media access control (MAC) and physical-layer (PHY) technology. It offers one gigabit per second (1 Gbit/s) raw bandwidth which is 10 times faster than Fast Ethernet (FE), by using a modified version of the ANSI X3T11 Fibre Channel standard physical layer (FC-0). Backward compatibility with existing Ethernet technologies is achieved by using the same IEEE 802.3 Ethernet frame format, and a compatible full or half duplex carrier sense multiple access/ collision detection (CSMA/CD) scheme scaled to gigabit speeds.

Gigabit Ethernet operates in either half-duplex or full-duplex mode. The full-duplex mode can be distinguished from half-duplex mode by the creation of a non shared point-to-point connection. This setup avoids the CSMA/CD access control mechanism and the transmission in which frames travel in both directions simultaneously over two channels on the same connection will be very efficient. Theoretically, this results in an aggregate bandwidth of twice

that of half-duplex mode. The half-duplex mode requires CSMA/CD methods and a channel can only transmit or receive at one time. A collision results when a frame sent from one end of the network collides with another frame. This potentially degrades Gigabit Ethernet's performance when operating in half-duplex mode.

2.3.1.2 MAC Flow Control

When Gigabit Ethernet operates in full duplex mode, it uses buffers to store incoming and outgoing data frames until the MAC layer has time to pass them higher up the legacy protocol stacks. During heavy traffic transmissions, the buffers may fill up with data faster than the MAC layer can process them. In this situation, flow control prevents the upper layers from sending until the buffer has room to store more frames. A possible loss of frames due to insufficient buffer space is therefore prevented. This flow control is also active when a receiving node is saturated with incoming data. When receive buffers approach their maximum capacity, a high water mark interrupts the MAC control of the receiving node and sends a control message to the sending node instructing it to halt packet transmission for a specified period of time until the buffer can catch up. The sending node stops packet transmission until the time interval is past or until it receives a new packet from the receiving node with a time interval of zero. It then resumes packet transmission. The high water mark ensures that enough buffer capacity remains to give the MAC time to inform the other devices to shut down the flow of data before the buffer capacity overflows. Similarly, there is a low water mark to notify the MAC control when there is enough open capacity in the buffer to restart the flow of incoming data. Full-duplex transmission can be deployed between ports on two switches, a workstation and a switch port, or between two workstations. Full-duplex connections cannot be used for shared-port connections, such as a repeater or hub port that connects multiple workstations. Thus, Gigabit Ethernet can be an effective interconnection network when running in full-duplex, point-to-point mode where full bandwidth is dedicated between two end-nodes.

2.3.1.3 Design Features

Gigabit Ethernet will eventually operate over a variety of cabling types. Initially, the Gigabit Ethernet specification supports multi-mode and single-mode optical fiber, and short haul copper cabling. Fiber is ideal for connectivity between switches and between a switch and high-speed server because

it can be extended to greater length than copper before signal attenuation becomes unacceptable and it is also more reliable than copper. These features however introduce much higher cost when setting up a large cluster network in comparison to copper cables. In June 1999, the Gigabit Ethernet standard was extended to incorporate category 5 unshielded twisted-pair (UTP) copper media. The first switches and network NICs using category 5 UTP became available at the end of 1999.

2.3.2 Scalable Coherent Interface (SCI)

2.3.2.1 Overview

Scalable Coherent Interface (SCI) [122] is not only a network interface card for message passing, but offers shared memory programming in a cluster environment as well. SCI intends to enable a large cache coherent system with many nodes. Besides its own private cache / memory, each node has an additional SCI cache for caching remote memory. Unfortunately, the caching of remote memory is not possible for PCI bus based systems. This is because transactions on the system bus are not visible on the PCI bus. Therefore an important feature defined in the SCI standard is not available on standard clusters and SCI is no longer coherent when relying solely on its hardware. An overview how SCI interfaces with current systems is given in figure 2.3. It may be noted, that this interface will look the same for other PCI network device interfaces.

In the current available SCI implementations, SCI does not offer message passing primitives. Instead, it exports and maps memory segments between different processes. The SCI network device will exchange data when the modification of memory is detect. For this, the memory is mapped according to figure 2.4.

2.3.2.2 Design Features

One of the key features of SCI is that by exporting and importing memory chunks, a shared memory programming style is adopted. Remote memory access (RMA) is directly supported at hardware level (Figure 2.4 depicts an overview of SCI address translations). By providing a unique handle to the exported memory (SCI Node ID, Chunk ID and Module ID) a remote host can import this 'window' and create a mapping. To exchange messages, data has to be copied into this region and will be transferred by the SCI card, which detects data changes automatically.

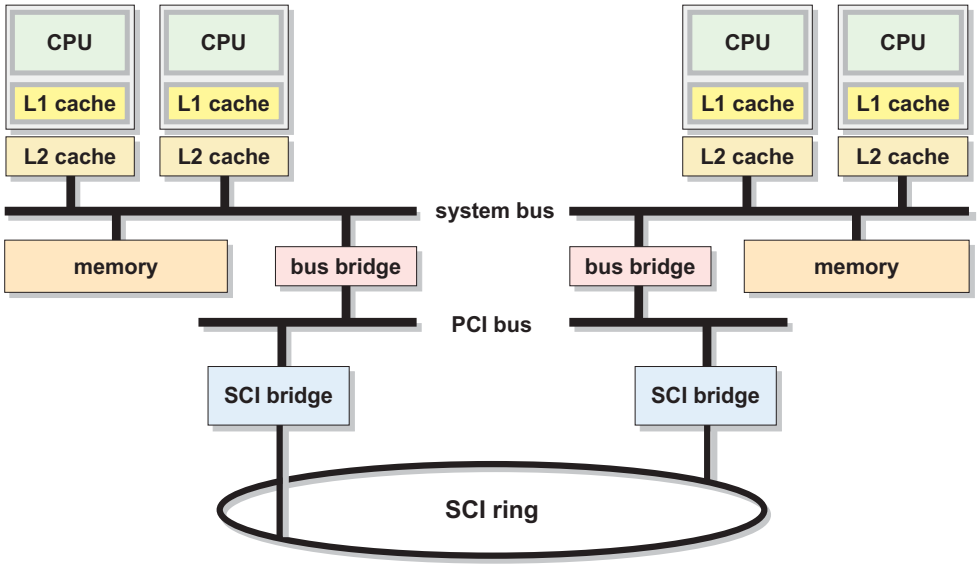

Figure 2.3: SCI Interfacing PCI Host Systems

2.3.2.3 SCI Functionality

Packet sizes of 64 Bytes are sent immediately, otherwise a store barrier has to be called to force a transaction. In order to notify other nodes when messages have been sent, they either can implement their own flow control and poll on data or create an interrupter which will trigger the remote host. However, the latter has a bad performance with a latency of 36 micro seconds on a Pentium II450. One major drawback of SCI is that a shared memory programming style can not easily be achieved because of the PCI bus lacks the functionality to cache regions of remote memory in the local processor cache. Furthermore, SCI uses read and write buffers to speed up communication which brings along a certain amount of inconsistency. Finally, SCI is not attractive to developers who have to keep in mind the big performance gap for read and write operations. For a Pentium II 450 the achieved performance was 74MBytes/s for remote write versus 7.5 MBytes/s for remote reads. When looking at concurrency then the preferred method is to use the processor to copy data. In this case however, the processor is busy and can not be used to overlap computation and communication as when DMA would be used. Using the processor, a remote communication in SCI takes place as just a part of a simple load or store opcode execution in a processor. Typically the remote address results in a cache miss, which causes the cache

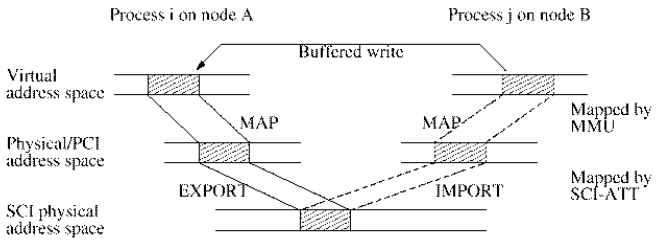

Figure 2.4: SCI Address Mapping

controller to address remote memory via SCI to get the data, and within the order of a microsecond the remote data is fetched to cache and the processor continues execution.

2.3.3 Myrinet

2.3.3.1 Overview

The Myrinet network is a highspeed interconnection technology for cluster computing. Figure 2.5 depicts the layout of the Myrinet NI. A complete network consists of three basic components: a switch, one or more Myrinet cards per host and cables which connect each card to the switch. The switch transfers variablelength packets concurrently at 2.5 Gbit/s using wormhole routing through the network. Hardware flow control via backpressure and inorder delivery is guaranteed. The NI card connects to the PCI bus of the host and holds three DMA engines, a custom programmable network controller called LANai and a minimum of 2 MByte of fast SRAM to buffer data.

2.3.3.2 Design Features

Under the supervision of the RISC, the DMA engines are responsible for handling data for the following interfaces: host memory/NICs SRAM and SRAM/network, respectively. In detail, one DMA engine moves data from host memory to SRAM and vice-versa, the second stores incoming messages from the network link to the SRAM, and the third injects data from SRAM into the network.

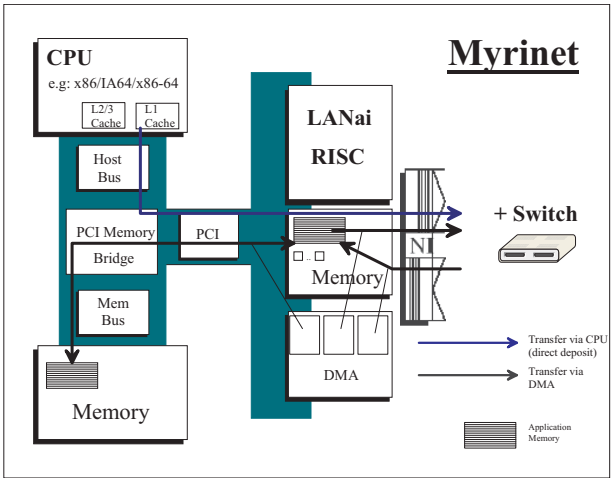

Figure 2.5: Myrinet

2.3.3.3 Myrinet Layout and Data Flow

The LANai 9 processor runs at 200 MHz, controls the DMA operations, and can be programmed individually by a Myrinet Control Program (MCP). The SRAM serves primarily for staging communication data, but also stores the code of the MCP. To simplify the software, the NI memory can be mapped into the host's virtual address space. [121] showed, that the limited amount of memory on the NIC is not a bottleneck, but the interaction of DMA engines and LANai. The Myrinet card retrieves five prioritized data streams into the SRAM. However, at a cycle of 10ns only 2 of them can be addressed whereas 3 are stalling. This leads to a stalling LANai, which does not get access to the staging memory. The effect becomes visible using clusters of 32 nodes or larger. When sending a message with Myrinet, first the user copies data to a buffer in host memory, which is accessible by the PCI bridge engine. A next step is to provide the MCP with the (physical) address of the buffer position. The LANai starts the PCI bridge engine to copy the data from host memory to NIC memory. Finally the LANai starts up the third DMA engine to inject the data from NIC memory into the network. On the receiving side, the procedure is vice versa. First, the LANai starts the receive DMA engine to copy the data to NIC memory and starts the PCI bridge engine to copy the data to an address in host memory (which was previously specified via a rendezvous protocol). Finally, after both copies are performed, the receiver LANai notifies the polling processor of the message arrival by setting a flag in host memory.

2.3.3.4 GM - Properties of the Myrinet Driver

Glenn's messages (GM) is a message-passing system for Myrinet networks. The GM system includes a driver, the Myrinet-interface control program, a network mapping program, and the GM API, library, and header files.

For application development at the user level, GM provides functions to allocate / release temporary buffers, that can be accessed by the DMA engines. Two types of buffers need to be created in order to exchange data. When sending smaller messages, the data is copied into send buffers which can be reused. It is not allowed to modify the content of these buffers until the message has been sent. GM provides mechanisms which indicate the successful delivery of messages. For this, GM acknowledges stored messages by a control message, which is sent back to the sender. The GM system will deliver this event to the upper software layers. As a consequence using this notification mechanism, the buffer can be assigned again for further transactions.

For larger messages, GM provides functions to register data to be accessible by DMA engines. In this case one data copy is avoided.

When receiving, the DMA engines spool incoming data into existing receive buffers. This has the advantage of clearing the NICs staging memory and provides the data to the application in user space.

For an application, the GM receive buffers appear as a single receive queue in which only the head and its payload can be extracted.

GM identifies a destination by a target node id and target port id. It does not require an explicit setup of a point to point connection. For this, it is required to have a mapper process evaluate the network. The GM system provides a total of eight accessible ports where three ports are reserved for the system itself. Thus, the mapper process is using one of the system ports to evaluate the network and each MCP will retrieve necessary routing information.

2.3.4 ATOLL

2.3.4.1 Overview

The ATOLL cluster interface network, is a future communication technology for building cost-effective and very efficient SANs with standard processing nodes. Due to an extremely low communication start-up time and very broad hardware support for processing messages, a much higher performance

standard in the communication of parallel programs is achieved. Four links of the interface network, an 8 x 8 crossbar and four independent host ports allow for creating diverse network topologies without additional external switches ('network on a chip'). This architecture especially supports SMP nodes by assigning multiple processes their dedicated device.

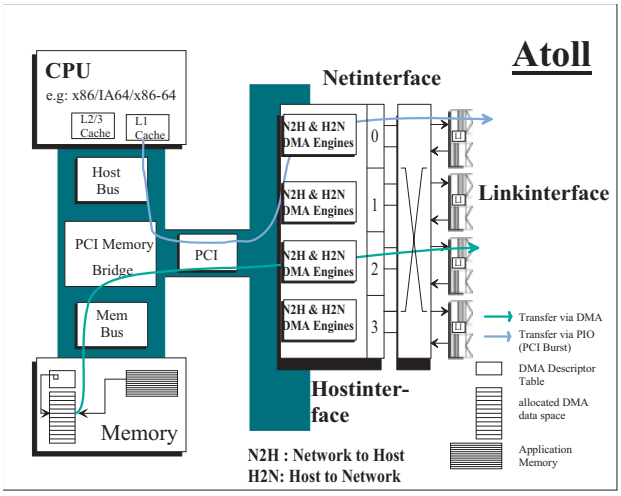

Figure 2.6: ATOLL - ATOmic Low Latency

2.3.4.2 Design Features

A special feature of ATOLL is the availability of multiple independent devices. ATOLL integrates four host and network interfaces an 8x8 crossbar and 4 link interfaces into one single ASIC. The card has an 64bit/66Mhz PCI-X interface with a theoretical bandwidth of 528MBytes/s at the PCI bridge. The crossbar has a fall through latency of 24ns and a capacity of 2GB/s bisection bandwidth. A message is broken down into 64Byte link packets, protected by CRC and retransmitted upon transmission errors. The chip itself, with crossbar, host- and network interfaces, is targeted to run at 250 Mhz. With a PLL frequencies can be adjusted.

2.3.4.3 ATOLL Hardware Layout and Data Flow

Standard techniques for the PCI bus such as write-combining and read-prefetching to increase performance are supported. Sending and receiving

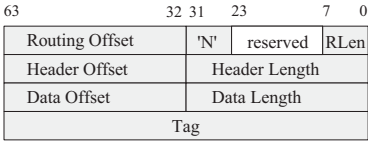

Figure 2.7: ATOLL Descriptor Layout

of messages can be done simultaneously without involving any additional controlling instances. The ATOLL API is responsible for direct communication with each of the network interfaces, giving the user complete control of "his" device. Thus supporting the user level communication concept. In contrast to other SANs, most of data flow control is directly implemented in hardware. Thus achieving a low communication latency of ¡ 2 micro seconds. ATOLL offers Programmed IO (PIO mode) and Direct Memory Access (DMA mode), respectively. A threshold value determines which method to choose. Routing is done via source path routing, identifying sender and receiver by a system wide unique identifier, the Atoll ID. Routing information is stored in a status page at the beginning of the pinned DMA memory space. For starting a transmission in DMA mode, a descriptor is generated and entered into the job queue of the host interface.

Injecting the message into the network is initiated by raising the descriptor write pointer, which triggers the Atoll card to fetch the message. Basically, the descriptor contains the following information: The message length, the destination id, a pointer to the message in DMA memory space and a message tag.

2.3.4.4 Atoll Hostinterface

The completion of a DMA task is signaled through writing a data word into main memory, which makes the time consuming interrupt handling by the processor unnecessary. Figure 5 depicts the three operations of a DMA send process. First, data is copied into the pinned DMA data space (1). Next, the descriptor is built and copied into the descriptor memory space (2). Finally, the write pointer (3) is raised, so that it points to the new descriptor. DMA data memory space and descriptor memory space are implemented as ring buffers. When receiving a message, the descriptor for the received message is assembled by the NI and copied into main memory. There it can be used cache coherently by the processor. If PIO mode is used for very short messages, the message is kept in the receive FIFO of the host

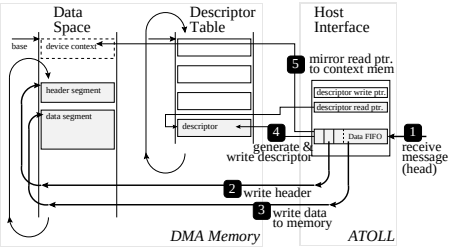

Figure 2.8: ATOLL Send Operation [89]

interface and the processor is informed of the received message through an update of the FIFOs entries in main memory. Just like in DMA mode, an expensive interrupt is avoided. To overcome deadlocks, a time barrier throws an interrupt forcing the processor to clear the network. In this mode, busy waiting of the processor on the FIFO entries leads to the extremely short receive latency. As this value is also mirrored cache-coherently into main memory the processor does not waste valuable memory or IO bandwidth.

2.3.4.5 Process of a DMA send job

The API creates a point-to-point connection for the communication of two processes. If two communication partners are within one SMP node, the ATOLL-API transparently maps the communication to shared memory. Last but not least, the demand for multi-threaded applications is supported with an additional special register which can be used as a semaphore 'test-and-set'. Unfortunately, nowadays processor's such as the PIII only allow locking mechanism at superuser level, thus achieving less performance as if the feature would be available in user mode.

Zero Copy mechanisms try to avoid any unnecessary data copies and are a focus of recent research projects. Two Copy DMA Basically, if PIO is available, then this communication mode can be used to directly inject data from user memory space into the network. On the receiving side, the message can again delivered directly. The disadvantage is that the processor will be involved during the entire transaction and can not be used for computation during that time. To enable the DMA engine to perform this task, a virtualtophysical address translation has to be implemented.

The TLB handling is usually performed by the OS. Basically, pages for translation have to be be pinned down, and virtual addresses now represent physical ones. The TLB can be placed and managed at the NI memory, the

host memory, or both. Using this method, Zerocopy can be achieved via remote memory write using the information provided with the TLB. Send and receive operations carry the physical address of the destination buffer and the DMA engine copies the data directly to the destination address. Typically, a rendezvous model is needed before such operation can take place, since the location at the receiver side is not know a priori. A limitation for this scheme is to have contiguous data. Also, the NIC has to support this functionality, being able to touch data which can be pinned down dynamically during runtime. This method also only makes sense, if the data which has to be transferred is locked down once and the region can be used. Otherwise expensive lock and unlock sequences will lower performance since a trap into the system will occur. Another problem coming along with zero copies is that of message flow control.

2.3.5 Infiniband

2.3.5.1 Introduction to Infiniband System Architecture

Over the recent years, several efforts were made to specify a new IO device which should overcome the drawbacks of the current PCI implementation. Two specifications Next Generation I/O (NGIO) and Future I/O eventually merged into SIO (08/31/1999) to be named Infiniband [33] afterwards shortly. The Infiniband Trade Association released the Infiniband Specification on 10/24/2000. Since then, Infiniband products have been presented on several plugfests. Infiniband Architecture (IBA) is a packet-switched System Area Network consisting of one or more IBA subnets interconnected via routers. In addition, a router may connect one of more IBA subnets to no-IBA environments such as Ethernet and Internet. Each IBA subnet may consist of one or more processor nodes (each containing one or more processors and memory arrays), IO Units (consisting of one or more IO controllers), IBA switches, and IBA and backplanes.

2.3.5.2 Infiniband Overview

Infiniband aims to become a high volume, industry standard I/O interconnect which extends the role of traditional in the box busses. For example other interconnects such as HyperTransport or Rapid I/O are currently implemented, but they will be used as an interconnect between processors only. Infiniband is unique in providing both, an in the box backplane solution and an external interconnect. In addition to internal interconnects, it provides

connectivity in a way previously reserved only for traditional networking interconnects. Therefore Infiniband introduced the notation of a Host Channel Adapter (HCA) which aims at interoperability with existing interfaces, Switches to connect several single host systems, routers to connect several subnets and Host Target Adapters (HTA), each of them having a Globally Unique Identifier (GUID), which will enable end systems (for example storage arrays) to be connected via Infiniband. Other characteristics are a 16 bit local id (LID) which is unique for each subnet, which is used to identify and route a packet to its destination in a subnet. Furthermore an IP address which is a globally unique 128 IPv6 Id used to identify an endnode by applications and to route packets between subnets.

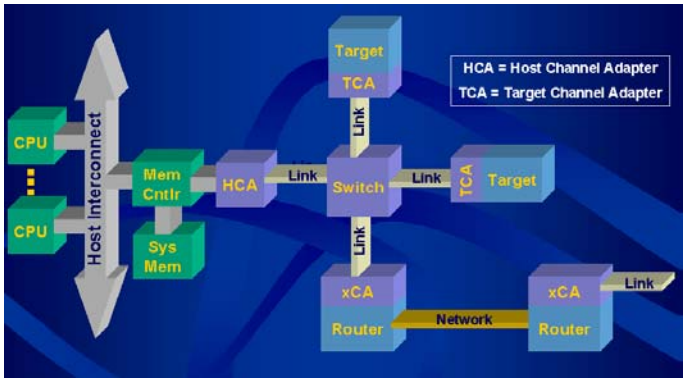

Figure 2.9: Infiniband System Overview [87]

Figure 2.9 depicts the detailed Infiniband Infrastructure ranging from Host Channel Adapters to Target Channel adapters. However, the Infiniband architecture is also designed to directly interface with the system bus. The IBM Summit chipset already has such an Infiniband port.

Figure 2.10 depicts a large Infiniband environment consisting of multiple subnets. The specification [33] determines 3 different rates at which Infiniband will operate. The 1x Infiniband will consist of 1 port running at 2.5Gbit/s. A 4x implementation will have 10Gbit/s. While 1x and 4x implementation are available today, the 12x implementation resulting in 30Gbit/s is expected to be introduced in 2004. Figure 2.12 depicts a comparison of Infiniband and other emerging specifications.

In which way Infiniband Layers exist and form up a transport protocol is depicted in Figure 2.11.

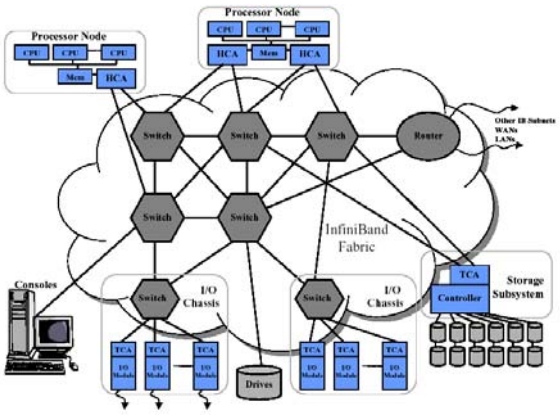

Figure 2.10: Infiniband System Overview [87]

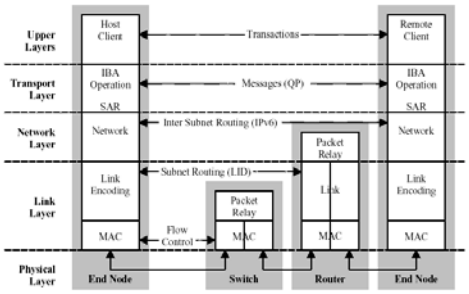

Figure 2.11: Infiniband Layers[87]

2.4 Performance

2.4.1 System vs User Level Mode Performance

Traditional Ethernet networks do not only come with a overburdened protocol stack, but also require system calls in order to send or receive messages. With U-Net [61], a paradigm was introduced which bypassed the kernel when communicating.

Feature	InfiniBand℠	PCI-X	Fibre Channel	1Gb & 10Gb Ethernet	Hyper-Transport™	Rapid I/O	3GIO
Bus/Link Bandwidth	2.5, 10, 30Gb/s[a]	8.51 Gb/s	1, 2.1Gb/s[b]	1 Gb, 10Gb	12.8, 25.6, 51.2 Gb/s[d]	16, 32Gb/s[c]	2.5, 5, 10, 20,... Gb/s[d]
Bus/Link Bandwidth (Full Duplex)	5, 20, 60Gb/s[a]	Half-Duplex	2.1, 4.2Gb/s[b]	2 Gb, 20Gb	25.6, 51.2, 102 Gb/s[d]	32, 64Gb/s[c]	5, 10, 20, 40,... Gb/s[d]
Pin Count	4, 16, 48[c]	90	4	4 (GbE); 8 (10GbE-XAUI)	55,103,197[a]	40/76[c]	4, 8,16, 32...
Transport Media	PCB, Copper & Fiber	PCB only	Copper and Fiber Cable	PCB, Copper & Fiber	PCB only	PCB only	PCB & connectors[b]
Max Signal Length PCB/Copper Cable	30in, 17m	inches	NA, 13M	20in, 100m	inches	inches	30in, NA
Maximum Signal Length Fiber	Km		Km	Km			

Figure 2.12: Infiniband in Comparison with Other Specifications[87]

2.4.2 Application Performance Enhancements through High Speed Networks

The replacement of massively parallel computers (MPPs) is an increasing trend over the recent years. The 19th edition of the Top500 list, a list which summarizes to 500 fastest systems according to their Linpack performance, held a fraction of 18% of clusters. For the 20th edition, two large commodity clusters are expected to be in the Top10 of best Linpack performance. When replacing an MPP with a cluster the most important topic to be evaluated is the interconnection network. As of today, two choices for a network exist in principal.

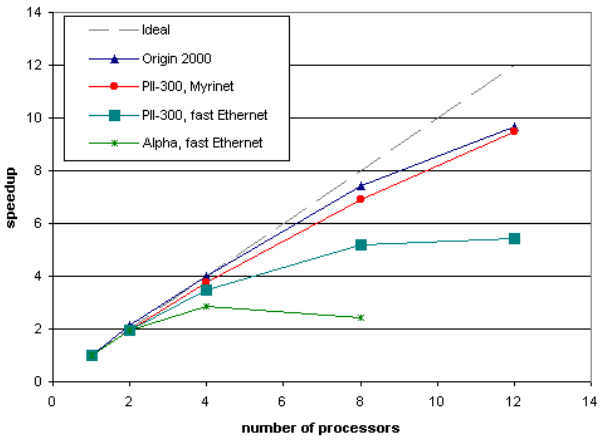

Figure 2.13: Hypersonice CFD Code Performance with respect to Machine Architecture and Interconnection Type[117]

One is to use a standard network such as Fast Ethernet, the other is

to use a high speed network or system area network. The latter roughly double the cost per node. This however only makes sense, if the set of applications which are going to be run on the machine can benefit from a fast network. This section will provide insight why a high speed network indeed does make sense. A large amount of applications is CFD or FEM code. The following graph shows comparative performance of a production CFD code (Hypersonic CFD code) developed at the University of Southampton on a variety of cluster architectures. The code solves the 2D Navier-Stokes equations for chemically-reacting flow. It uses MPI communications in a pipeline topology and has been in production on SP2 and Origin 2000 systems for several years. In this graph the performance on three different Windows NT cluster configurations is shown, together with comparison with an Origin 2000 dedicated machine with 225MHz R10000 processors. True speedup, relative to a purely sequential, single processor version is shown.

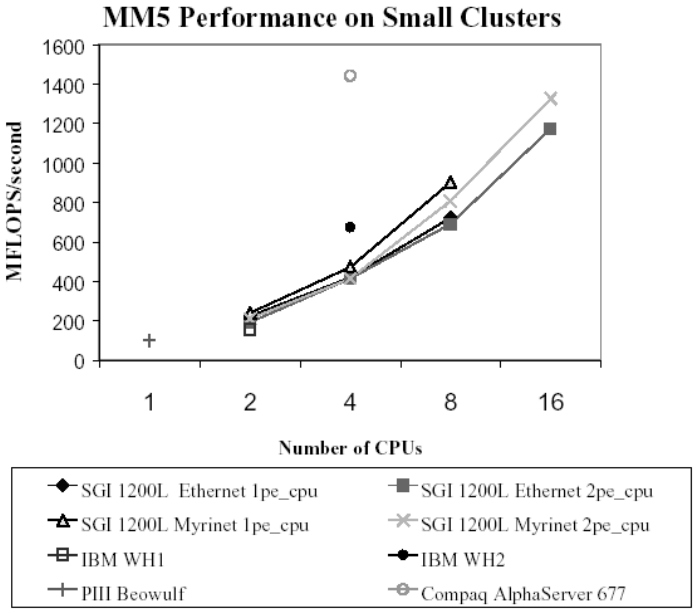

Figure 2.14: MM5 Speedup in Comparison using Myrinet or Fast Ethernet as Interconnection Network. [116]

However, a fast network is not always required for parallel applications. Especially master slave applications for example in which data is only send forth and back between master and slave but no communication is performed among the slaves will have a better price performance ratio when using tradi-

tional networks. Figure 2.14 however is a distributed application which also achieves good scalability. For this application, a high speed network would not be required and a traditional network is sufficient.

Chapter 3

Extending the Parallel Virtual Machine with a System Area Network Plugin

With the variety of different networking devices and multiple low level API's for the same device (GM, PM and BIP for Myrinet for example), it would be efficient to provide a specified generic API for a higher message passing environment to allow for an easy integration of existing or new interconnects.

The portable MPI implementation MPICH [103], which provides a Chameleon device, is an approach to incorporate multiple devices using an abstract device interface (ADI).

Another message passing environment is the Parallel Virtual Machine (PVM) [108], which provides more flexibility than current MPI implementations. Some of its advanced features are dynamic creation of processes and communication partners, or fault tolerance. But also a collection of very heterogeneous machines can be harnessed to form one single entity, transparent to the application through the PVM message passing library.

The MPICH implementation of MPI provides an abstract device for communication layers and several channel devices have been implemented.

Contrary to MPICH, PVM does not provide an abstracting device interface, but has a mixed code that differentiates between a variety of architectures, ranging from MPP nodes to a network of workstations. For workstations, control is given by additional PVM daemons (PVMd), which are running on each host of the virtual machine. Another part of the PVM system is the PVM library (*libpvm*) to which tasks have to be linked. Tasks in the PVM system are represented by a unique task identifier (tid). Messages

are sent using the *tid* as destination parameter. Within PVM, two routing policies are given. Using the default routing, messages are transferred via the UDP connected daemons which route the message to the final destination.

To improve communication performance a direct connection between two tasks can be established, leaving the PVMd's outside the transfer. This mechanism has been used to extend the PVM communication primitives to provide an interface for other network devices not using TCP/IP protocols.

Figure 3.1 depicts the logical PVM layout in which PVM daemons and tasks communicate over a variety of platforms. The goal is to replace the heavy TCP/IP protocol stack with a low level, reliable system area network transport.

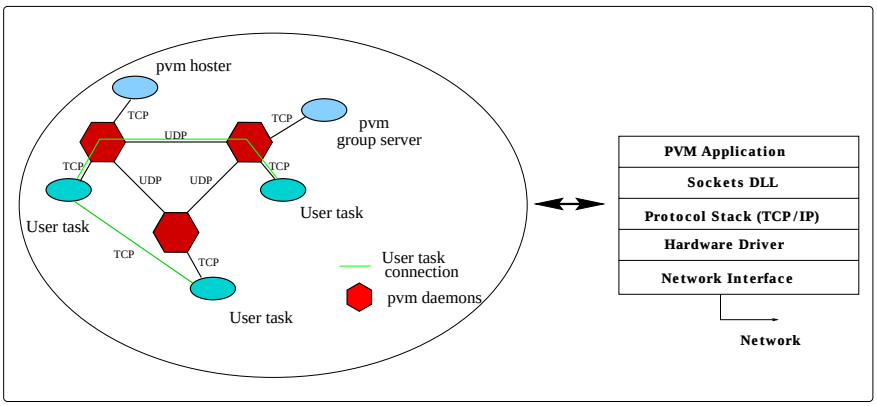

Figure 3.1: PVM Communication Overview using Ethernet. PVM daemons are connected through the connection less UDP protocol to allow for large virtual machines. Tasks are connected to the daemons through the connection oriented TCP protocol. Tasks can create direct connections to other tasks, otherwise the messages are routed through the PVM daemons.

This point to point design would also fit most of current system area network implementations very well.

Other research used ATM as a network [88], still relying on TCP/IP and not gaining much more performance than using Fast Ethernet. Also, multi devices which handle several network connections were not supported, but are addressed within this extension to the standard PVM. Finally, other ports to SCI are available [107], [106].

3.1 A Common Interface for a SAN Extension to PVM

For a message passing environment like PVM, several functions are required in order to establish a point to point connection between two tasks. One important feature for a dynamic environment is to allow the establishment of connections during runtime. Obviously, this feature is also resource friendly since it does not set up unnecessary connections per se, which are subject of not being used during execution. In order to allow this, a request for setting up a direct connection must be transported from one node to another. The initial request structure typically includes basic SAN specific information.

The delivery of requests is performed using an additional standard network such as Fast Ethernet. The standard PVM provides controlling daemons which will deliver data between tasks until a direct connection between two nodes using a faster method is made.

To make this scheme universal to several interconnects, this transfer is split up in a request and a ack/grant phase between two nodes (see figure 3.2 for details).

Host Id, Chunk Id and Module Id need to be exchanged in order to provide a direct point to point connection. In order to set up a bidirectional connection, (1) will provide the Sender's Request via Daemon A and B to task B. In (2) the Receiver will reply necessary information from the destination. With (3a) and (3b) two different windows will establish round robin buffers into which the actual data can be delivered.

This mechanism can be split up in the following phases a

1. Request - a node invokes a new pt2pt connection, passes the request to the daemon and waits for an acknowledgment after which the connection is finalized through a reply from the remote target.

2. Delivery - a controlling daemon holds the request and the data is passed via conventional communication mechanisms to the receiver

3. Ack and Grant - a node receives the request when it enters a library function, performs steps to setup and finalize the connect and returns a ACK if resources are available

After this exchange of SAN information further communication takes place only using the fastest interconnect available. Typical content of a setup message can be a port number, node id, or memory addresses.

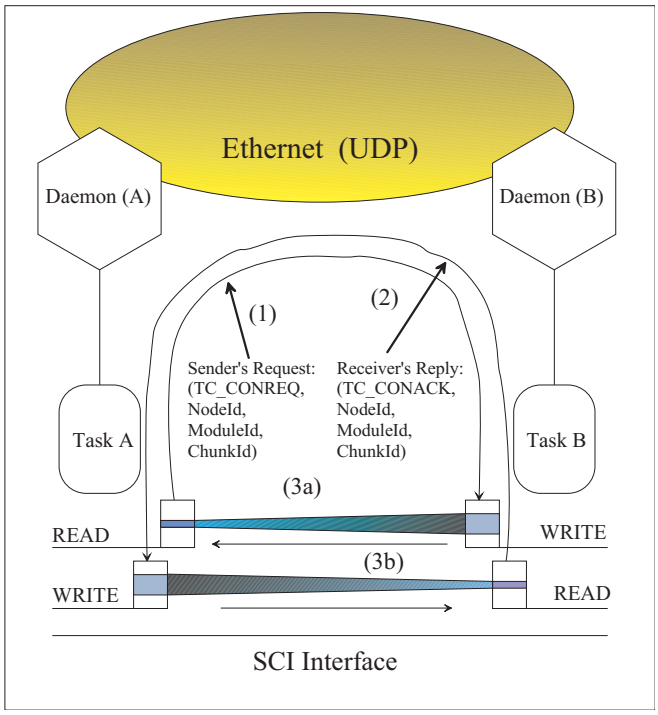

Figure 3.2: Establishing a direct SAN connection between two Nodes. In this example,the request and grant protocol is presented for the SCI network

.

When a direct communication is established, low level calls to the API of the interconnect can then be used to exchange data. For this, a sender holds a struct for each communication partner in which the type of connection is stored. However, for a receiving node, message arrival has to be detected (and received) from different networking devices.

3.2 Implementations on Different Interconnect Devices

In the following the extensions to PVM are explained which allow the plugin of different interconnects. To allow a common interface for several interconnects, a plugin has to register its functions in a header file which is included into the lpvm.c file which implements the setup of new connections as well as send/recv functions and detects message arrival. As described above,

the following functions are prototyped *san_request(), san_ack(), san_send(), san_recv() and san_poll().*

When enabling direct connections, a PlugIn references and calls the registered functions (e.g: the request and ack, the send/recv and select functions) from the header file. For them it is required to be autonomous functions not requiring additional control (for example the *request* function provides **all** required information for the receiver, or a *send* function does only return when a (possibly later) message delivery can be guaranteed.) Thus, the extended PVM does not differentiate between various devices but calls the registered functions in the same manner. In order to detect a plugin, the extended PVM tries to load a shared library. If available, function will get registered.

3.3 Plugin Implementation Details

3.3.1 The PVM-SCI plugin

Scalable Coherent Interface (SCI) is the international standard IEEE #1596-1992 for computer-bus-like services on a ring-based network [109]. It supports distributed shared memory (DSM) and message passing for loosely and tightly coupled systems. Optionally cache coherent transactions are supported to implement CC-NUMA (Cache Coherent Non-Uniform Memory Access) architectures. However, using the PCI bus as commodity interface to build a SAN, major key features of SCI, such as the cache coherency, no longer exist since it is not possible to snoop on the host memory bus. Thus, for each message transaction, data has to be explicitely memcopy'd into memory mapped regions.

Another disadvantage is the performance difference for the so called *put* or *get* scenario. Writing to the network is an order of magnitude faster than reading from the net work (*writing to* or *reading from* remote memory. In our environment these effects compared with 73MBytes/s to 12MBytes/s respectively. This must be kept in mind by developing and implementing an efficient message passing plugin for a SCI network.

3.3.1.1 Details on PlugIn functions

When establishing a direct connection, a basic information is the SCI-node ID, the SCI memory location and the chunk id to map exported memory. This setup is performed by the initiator **and** the recipient, both exporting

to (during setup) and mapping from (when finalizing the direct connection) the communication partner (the remote task). Thus, the initiator receives necessary information within the ACK/GRANT reply.

- For send functionality, a ring buffer has to be implemented. This is required since SCI only exports memory but does not offer functions useful for message passing. Data is transferred by copying data into exported memory regions. The SCI card will detect these changes and transport it to the node which has mapped the memory region. The ring buffer has been implemented as depicted in figure 3.3.

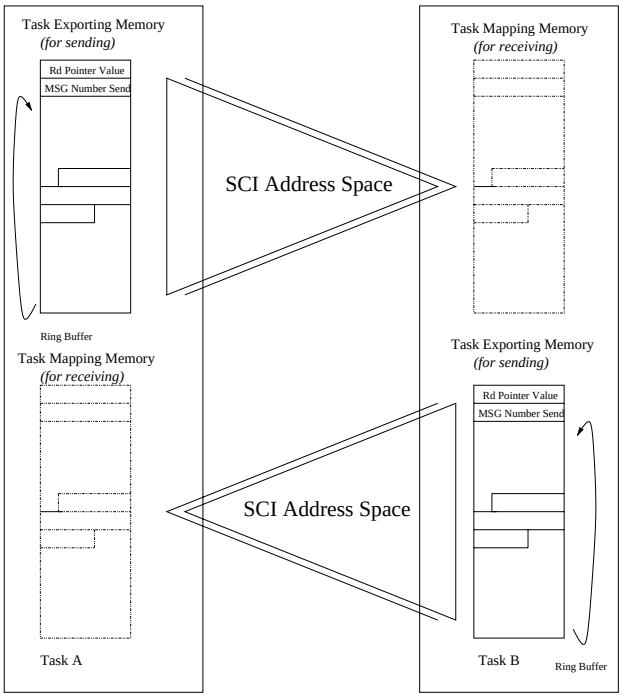

Figure 3.3: Ring buffer implementation for the PVM-SCI plugin. Each communication partner exports a chunk of memory in which data can be written. The chunk is separated by a small header which holds necessary flow control information and a data region which holds messages. Messages themselves contain an envelope or header and the corresponding payload.

The remote side updates a read pointer indicating the last position of the ring buffer read. This way, a sending process does not overwrite unread data and flow control is guaranteed. An earlier version of the

plugin used SCI's remote interrupters, however the performance (it took about 60 micro seconds) was too slow to achieve any performance. When a sent has been made, the number of messages sent is updated on the remote side [106].

- For recv functionality, a ring buffer is implemented as well. After receiving a message the read pointer of the last position on the remote side is updated. As a consequence, two nodes export memory which is mapped by the communication partner.

- select/poll functionality is implemented by querying the number of messages received for each communication partner. A local structure stores the numbers read so far and if for any communication partner this number is higher, the function returns with a pointer to the new message. The plugin also used the SCI interrupter mechanism. This lets a node raise an interrupt on a remote host. However, the performance using SCI interrupters was less than using Fast Ethernet. To achieve efficiency, the interrupter method could not be used since context switches hampered the performance. Also the notification mechanisms had to keep in mind the enormous performance differences for the put and get scenario. Thus, for example, the *number of messages* sent is stored locally at receiver side, while the *read* pointer is stored in physical memory of the sender.

3.3.2 The PVM-GM plugin

Myrinet [9] is a high speed interconnection technology made by Myricom. It uses source path routing and is capable to transfer variable message lengths at 2 Gbit/s. The (open source) GM driver including a Myrinet Control Program comes from Myricom and implements DMA only. GM provides a connection less protocol in which sender and receiver are identified by so called host id's and GM ports. A process gets access to the Myrinet by opening a GM port. However in order to send or receive data, a process must provide a pool of receive buffers (of different) sizes which can be accessed by Myrinet's DMA engines. First, data is then transfered from host memory to network memory (SRAM on Myrinet card), then injected into the network. Obviously incoming data should be transfered to host memory as soon as possible clearing the network and the limited SRAM memory on the Myrinet card. Thus the plugin has to provide enough receive buffers so that the DMA engines always find a slot into which the data from the network can be stored.

3.3.2.1 Details on Plugin functions

When establishing a direct connection, GM port information is exchanged, but an establishment of a connection is not necessary. The following list will provide detailed information on how the necessary plugin functions can be implemented based on the GM API.

- For implementing the send function, the `gm_send_with_callback()` mechanisms which are already provided by GM can be used. For a message transfer, data is first copied into an appropriate bin. To achieve better performance, this data is pipelined (see section 3.3.5 for details). One key feature is the flow control coming with GM. In particular, a function can be given to the `gm_send_with_callback()` function which is called after the send has completed. This is of importance since with the efficient usability of DMA's, overlapping of computation and communication can be implemented easily. A bin can then be re-used when the function has been called, signaling the end of the message transfer.
- Implementing the recv function for the PlugIn is also straight forward by using the `gm_receive()` function which returns the first entry from a FIFO queue. Into this queue all incoming messages are inserted and when finding an entry in the queue message data already has been DMA'd from network memory to host memory into one of the provided bins. Further GM functions provide the source of the message to be identified for higher level message passing systems. That is that every event is passed to the upper layers as a structure which holds the origins GM node id and port id. It also contains additional information such as the message length for example.
- A select/poll mechanism is provided as well through GM. The `gm_receive()` function returns a NO EVENT tag, if no message has arrived or directly provides the message. Thus, the function is non blocking by default.

3.3.3 PM2 Plugin for Myrinet using the SCore Environment

PM2 is another low level API for Myrinet. It is developed by the Real World Computing Partnership (RWCP) Japan [98] and is open source. PM2 can be used as a low level programming API, but is specifically designed to run under the SCore environment. This environment can be seen as a

cluster operating system. Its programming models include a MPICH-PM device, a tuned Ethernet over PM implementation as well as an OpenMP implementation. Internally, SCore implements gang scheduling in order to increase the efficiency of parallel applications. Under the supervision of the SCore daemons, the PM driver is capable of autonomous checkpoints. The SCore daemons, being a central instance of control, required additional efforts for an integration into PVM. The following picture depicts the resulting control layers.

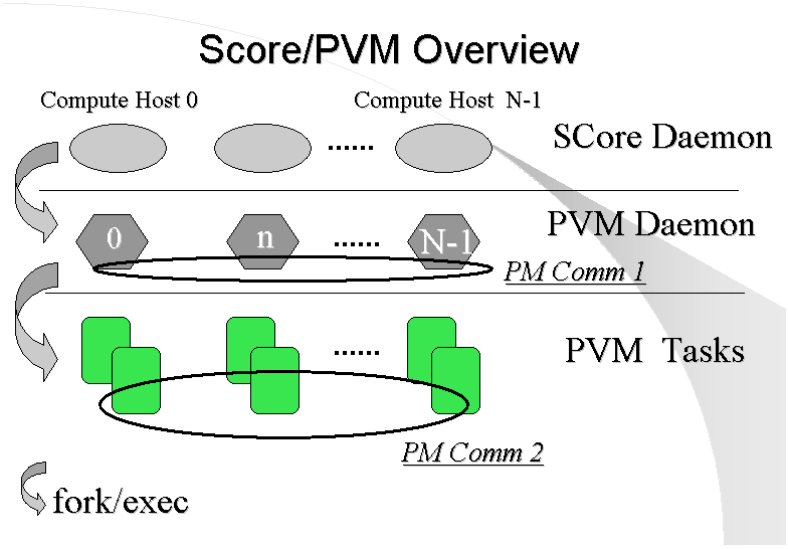

Figure 3.4: Overview on Different PM Context Layers

Basically, the SCore daemons will start applications passing an optimized routing table to the parallel application. This routing information is calculated in dependency of other applications. A restriction is that only N processes can be used within a session. Adding further nodes is not possible which is a burden for PVM or MPI-2 applications where the programming model offers support for dynamic process management. As a solution, each SCore daemon first forks the controlling PVM daemons. These daemons communicate within their own PM context. The actual application is then started using a new PM context. As a consequence, a user has to specify the maximum amount of processes to be used. These additional nodes are then a resource pool for further tasks to be spawned by PVM.

A program using PM2 must first open the PM device and then open a context which has to be bound to a channel. Buffers are provided within

PM2 functions.

A process calls the `pmGetSelf()` function to identify itself in the PM context. Other nodes may use this information to send messages. A key feature of PM2 is the availability for different platforms. Not only Myrinet is supported, but Fast Ethernet and Gigabit Ethernet as well. This way, also for conventional interconnects, the overburdened TCP/IP protocol has been replace by using a modified version of UDP with additional flow control. The resulting performance is much better than when using TCP/IP [99]. Moreover, the problem of limited numbers of file descriptors which would cause larger Ethernet clusters to fail using TCP/IP has been solved. This feature allows to run MPICH applications on hundreds of nodes using Ethernet, since the protocol is connectionless.

3.3.3.1 Details on Plugin functions

When establishing a direct connection, the PM port number information (0, .. , n-1) is exchanged but like GM, a setup of a connection is not necessary. The following list reflects the required implementation details when using the PM as the low level protocol.

- For implementing the send function, the `pmSend()` function can be used. It uses only an internal context value as a parameter, which has been initialized by PM when providing a send buffer which was tagged with the destination node information.
- The Recv Function can be easily implemented with the `pmReceive()` function.
- Select/Poll - Like GM, the behavior within PM2 is the same, the `pmReceive()` function is non blocking.

3.3.4 Optimized Memcpy Functions

Recent research focused on zero copy message transfers which avoids unnecessary copies of data. This often requires a lot of changes to higher level code, but does not result in dramatic gain of performance (2-5 per cent [100]). When developing different PlugIn's for different network interconnects, it became clear, that for each selected host architecture optimized memcpy functions can increase the performance compared to C lib memcpy function calls. A major gain can be seen when using SCI. Here a bandwidth limiting factor

is the performance with which data is copied to exported memory. Using the standard memcpy call the performance peaked at only 22MBytes/s. the standard memcpy call the performance peaked at only 22MBytes/s. With optimized memcpy routines (for example using the FPU‘s 64bit operands) the peak value achieved on a PII 450 system was 73 MBytes/s, compared to 280MBytes/s when copying data from shared memory to local memory. Thus the maximal performance for higher level message passing environments such as PVM or MPI is limited by the put performance of 73MBytes/s. Further optimizations such as MMX, SSE, SIMD or VIS can be considered as well.

3.3.5 Data Pipelining

When using DMA capable hardware such as the Myrinet network card, a major performance increase can be made by pipelining / interleaving message data. In this case, separating larger messages into multiple chunks, which can be chained by the DMA, increase the performance, since a memcpy of a shorter message takes less time and a first DMA transaction can already start while other memcpy's follow. This simple technique for example increased the performance for transferring larger amounts of data from 32 MBytes/s to 79 MBytes/s.

3.3.6 Memory Registration for Direct Transfers

To avoid data copies into an extra buffer some operating systems like LINUX provide the functionality of registering memory. In this case data is pinned down and can be accessed by DMA to be injected into the network directly. This enables an optimization for some operating systems to switch between different communication patterns depending on the message size. Registering memory however involves a costly kernel trap. In particular, on a PIII 600 with Linux 2.2.14, we measured a break even of only 11KBytes, so that a memcpy of data performed less than registering memory. Thus, for larger messages better performance was achieved by switching the message protocols.

3.4 Performance Comparison for Different Plug Ins

In this section the performance numbers using the `nntime` program, which comes with the PVM distribution are presented. The nntime program mea-

sures the round trip performance. A message is sent to the destination, it is received and consumed and another message is assembled and send back. The payload for these data transfers is the same. This test is then conducted for a sample rate which can be specified. For the following tests, a sample rate of 100 iterations was used. Each measurement is compared with the *raw* performance of the used low level API. With raw performance we mean the potential performance which is available when using the low level API directly without any further overhead involved. Using a low level API downgrades portability while using a portable message passing interface like PVM performance is sacrificed.

3.4.1 PVM-SISCI Performance

Figure 3.5 depicts the nntime benchmark performance using the SCI network. The plugin uses the SISCI low level API from Dolphin. When developing the plugin, the code was instrumented to see potential performance losses. It was immediately visible, that only optimized memcpy routines which copy the message into the SCI address space are required. Each CPU architecture has the potential to be tuned without providing portability. Thus a memcpy routine for a Pentium II would no longer work or be efficient on Pentium III processors for example. Since this work focused on efficient communication layers, the investigation into providing efficient copy routines was neglected. Optimized memcpy routines have been used from the Yasmin project [123]. The tests were conducted on a hpcLine system at the Paderborn Center for Parallel Computing. When conducting the development and measurements, the system was equipped with Dual Pentium II 450Mhz nodes using an Intel 440 GX chipset. The system contained 4th generation PCI/SCI adapter cards (D308 revision D) that have been designed by Scali AS. These cards have an 32bit/33Mhz PCI interface and are based on Dolphin's CluStar technology (PCI/SCI card 310).

The performance analysis reveals that especially for small messages most of the raw performance is lost due to the overhead increased through PVM. Nevertheless, this plugin provides a much lower latency when compared with Fast Ethernet. With a peak of 49.2 MBytes/s and a one way latency of 45 micro seconds the PVM-SCI plugin outperforms Fast Ethernet which provides a latency of 140 micro seconds and a maximum bandwidth of 8.8MBytes/s.

With an upgraded system (the PII 450 processors were replaced by PIII 850 Mhz processors), a highly optimized MPI implementation from Scali shows a maximum bandwidth of 85 MBytes/s and a latency of 5.1 micro seconds. This improved performance is partly due to the new PIII processor

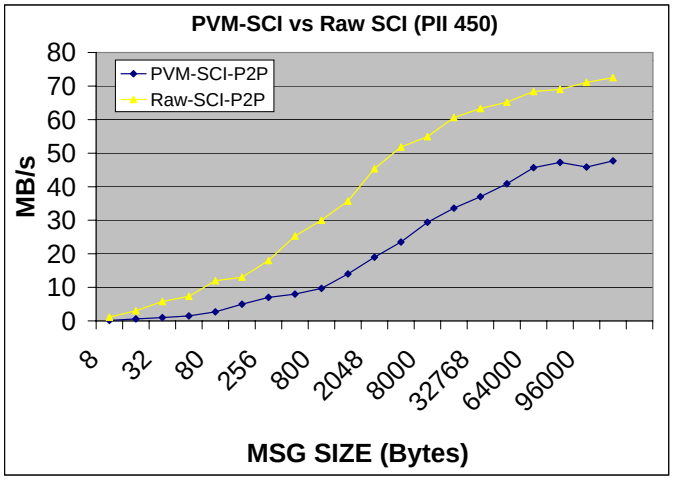

Figure 3.5: nntime Performance under PVM-SCI. For small messages performance is lost due to PVM's feature to allow a heterogenous interconnection network. In this case nodes inside a cluster can communicate efficiently, but also connection to external nodes can be established. This is ideal for Grid application which will make use of fast networks when available, reverting to the traditional protocol otherwise.

which is capable of providing higher memcpy bandwidth. Most of the improvement however can be found by analyzing the message passing protocol. PVM remains flexible allowing additional communication to tasks which are out of the System Area Network. This means that under PVM the library also checks for requests for establishing new connections or other control messages through the PVM daemon. This check however involves a system trap by calling the `select` function. As a consequence, the latency using multiple devices including a trap into the system can not be as low as the latency of a message passing implementation which only uses static communication patterns. The implementation was also extended by using pthreads to listen on Ethernet communication. However, a significant performance gain could not be achieved due to the high cost of thread schedules.

3.4.2 PVM-GM Performance

Figure 3.6 depicts the nntime benchmark performance using the GM low level protocol running over the Myrinet 2000 network. The setup consisted

of two dual Pentium III 600 Mhz nodes using an Intel LX chipset. The PCI interface was 32bit/33Mhz. The data transfers are using DMA engines. The PVM-GM plugin sacrifices performance for smaller messages. It peaks with 72.2 MBytes/s and has a latency of 50.2 micro seconds. These values are achieved by using the pipelining strategy in which the message is fragmented to 16KBytes to overlap with DMA operations.

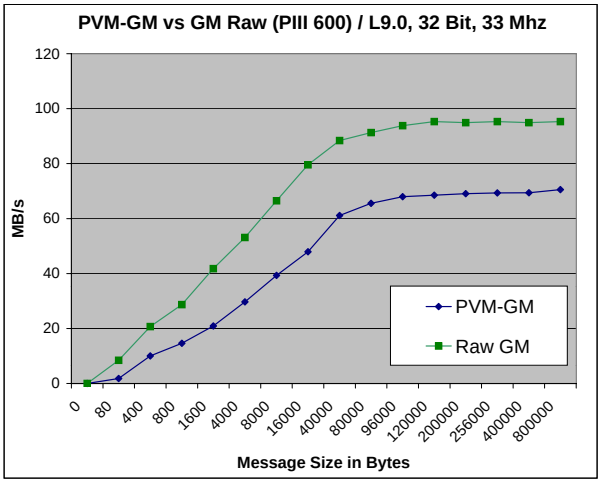

Figure 3.6: nntime Performance under PVM-GM.

3.4.3 PVM-PM Performance

Figure 3.7 depicts the nntime benchmark performance using the PM driver over Myrinet 2000. The test was conducted at the SCore III cluster hosted by the Real World Computation Partnership (RWCP). The system consists of 512 dual PIII 933 Mhz nodes using a Serverworks LE chipset. The Myrinet 2000 network cards were plugged into a PCI 64bit/33Mhz bus. The PM low level API is an optimistic protocol available for Linux operating systems only. Some of the notification mechanisms are better implemented than the GM driver.

This optimistic protocol as well as the doubled PCI bus performance using 64bit allow the PVM plugin to gain further performance. The PM interface also has a built-in shared memory interface. This means that two processes communicating on the same node will use shared memory for communication, still using the PM device. Both performance measurements are presented.

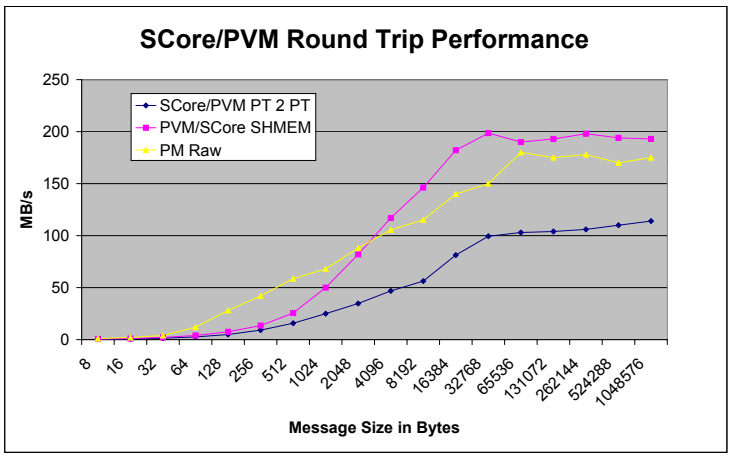

Figure 3.7: nntime Performance under PVM-PM. The fast Serverworks LE chipset with a 64bit/33Mhz let PVM reach 100+ MBytes/s. PM internally provides a shared memory interface as well. Without crossing the PCI interface, the performance can be further increased.

3.4.3.1 Conclusion on Performance Results

The Parallel Virtual Machine has been extended with a plugin which allows the integration of reliable low level transports. All PVM features are still provided. This partly limits the plugin to exploit the potential performance given through the low level API. However, compatibility was the major concern, since some applications do not fully benefit from much faster networks.

Latency sensitive applications will benefit from reduced communication overhead as for some networks like Myrinet, the DMA engines will transfer the data out of registered buffers. The application however can continue processing data.

The nntime benchmark implements a ping pong model in which applications send out a request and wait for a response. This is different from the ping ping model in which a sender continuously streams data into the network. For the latter message, final message as an acknowledgment from the destination then ends this performance measurements. Values for the ping ping model are typically much higher since overhead for processing the message at the receive side is neglected.

Chapter 4

Communication Environments for the ATOLL Network

The ATOLL network provides a low level API which is designed to support message passing environments efficiently. This chapter describes the integration of the ATOLL network into existing environments.

4.1 The MPI Environment for the ATOLL Network

In this section the adaption of an open source reference implementation of MPI to run over the ATOLL network is presented.

4.1.1 A MPI Reference Implementation: MPICH

The MPI specification effort started in 1992 and MPICH was an immediate implementation as the specification evolved. The early MPICH was improved by providing better performance and support for all required functions specified by the MPI standard. Furthermore, the Abstract Device Interface (ADI) architecture was introduced. This attracted individual vendors and other developer to take advantage of this interface to develop their own highly specialized implementations of it. As a result, extremely efficient implementations of MPI exist on a variety of machines. In particular, Convex, Intel, SGI, and Meiko have produced implementations of the ADI that produce excellent performance on their own machines, while taking advantage of the

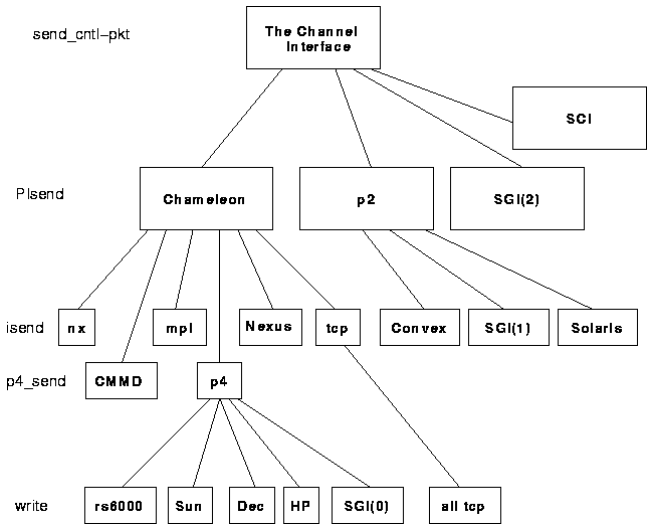

Figure 4.1: Overview of the MPICH Channel Interface [124]

portability of the great majority of the code in MPICH above the ADI layer. Figure 4.1 depicts available ADI implementations.

This approach also has been followed when developing the ATOLL ADI implementation.

4.1.2 MPICH ATOLL Device

In the following, some implementation details for the ATOLL network integration into the portable MPI implementation MPICH will be given. It addresses how a parallel application will be started and how the low level ATOLL API has been integrated into the abstract device interface.

4.1.2.1 Process Placement

MPICH offers a two methods for application startup. First is to use a central process which starts remote processes through rsh or ssh. Second is the process startup through additional daemons which are running under the root account on every host. Through the `setuid` command, a fork'ed child is placed under the actual user account. The daemon method is more efficient for very large clusters, since the remote procedure calls are more costly. We have focused on using the traditional startup which works as follows. Using

the mpirun script one process starts other processes on remote nodes. These nodes are specified by a machinefile. Upon process startup, the processes first enroll into ATOLL and receive their `atoll_id`. This startup mechanism offers several advantages. One is that a socket is created with the central instance and messages can be exchanged. This way, the remote processes pass back their atoll_id. Moreover, the central instance will detect applications experiencing an error (e.g: SEGV) through an EOF on the socket. This way, all other process on remote nodes can be informed to shut down, which is how the MPI standard is specified. After receiving all atoll_id's from the remote processes, it can be guaranteed, that the parallel application is able to use the ATOLL network for communication. To finalize this setup procedure, the central instance will distribute a table of matching MPI node id's ranging from 0, .., n-1 and their corresponding atoll_id's to all processes involved in the parallel application. This way, each process can build a connection by referencing the requested MPI id.

4.1.2.2 Protocol Adaptions

The ATOLL network hardware implementation has been designed very efficiently. This includes that messages have to be a multiple of 64bit. A message in MPICH consists of a header to identify a message and a payload. The header struct is easy to adapt to be a multiple of 64bit, however the payload has to be padded. This can be done in a brute force manner which would degrade performance by splitting up or extending the message. We have chosen to modify the ATOLL low level API to support any message size. That is the actual message which is not restricted to be a multiple of 64bit will be transferred. If a padding is needed, then only the descriptor is modified, letting the hardware transfer 1, .., 7 Bytes more than needed. This solution is therefore much more efficient than the software approach.

4.1.2.3 The ATOLL Channel Interface

The central mechanism for achieving the goals of portability and performance is a specification which is known as the abstract device interface (ADI) [91]. All MPI functions are implemented in terms of the macros and functions that make up the ADI. All such code is portable. Hence, MPICH contains many implementations of the ADI, which provide portability, ease of implementation, and an incremental approach to trading portability for performance. One implementation of the ADI is in terms of a lower level (yet still portable) interface which is called the channel interface [92]. The channel interface can

be extremely small (five functions at minimum) and provides the quickest way to port MPICH to a new environment. Such a port can then be expanded gradually to include specialized implementation of more of the ADI functionality. The architectural decisions in MPICH are those that relegate the implementation of various functions to the channel interface, the ADI, or the application programmer interface (API), which in our case is MPI. At the lowest level, what is really needed is just a way to transfer data, possibly in small amounts, from one process's address space to another's. Although many implementations are possible, the specification can be done with a small number of definitions. The channel interface, described in more detail in [92], consists of only five required functions. Three routines send and receive envelope (or control) information: MPID_SendControl,One can use MPID_SendControlBlock instead of or along with MPID_SendControl. It can be more efficient to use the blocking version for implementing blocking calls. MPID_RecvAnyControl, and MPID_ControlMsgAvail; two routines send and receive data: MPID_SendChannel and MPID_RecvFromChannel. Others, which might be available in specially optimized implementations, are defined and used when certain macros are defined that signal that they are available. These include various forms of blocking and nonblocking operations for both envelopes and data.

These operations are based on a simple capability to send data from one process to another process. No more functionality is required than what is provided by Unix in the select, read, and write operations. The ADI code uses these simple operations to provide the operations, such as MPID_Post_recv, that are used by the MPI implementation. The issue of buffering is a difficult one. We could have defined an interface that assumed no buffering, requiring the ADI that calls this interface to perform the necessary buffer management and flow control. The rationale for not making this choice is that many of the systems used for implementing the interface defined here do maintain their own internal buffers and flow controls, and implementing another layer of buffer management would impose an unnecessary performance penalty.

4.1.2.4 MPICH/ATOLL Protocols

To handle different message lengths, different internal MPICH protocols can be implemented. Their importance is explained in the following. One impact of a message length is that of network congestion. Assumed, a message is not limited in its size by hardware, long data transfers will block paths making it impossible for other messages to get through. ATOLL, specifically, does not have such a Maximum Transfer Unit (MTU). Another reason for protocols

is the possibility of pipelining fragments of messages and therefore speeding up the performance because the first chunk of data can be already handled by the NIC, while other fragments are prepared by the host processor. In this case, a CPU only spools a fraction of the message into the SEND DMA region and assembles a descriptor before spooling another fraction of the message. As a consequence, the ATOLL hardware is triggered and fractions of the message are already on its way to the destination. This pipelining strategy has been proven to increase the performance (Reference: PVM-GM with and without Pipelining).

Therefore, the MPICH/ATOLL device uses three different protocols: SHORT, EAGER and RENDEZVOUS. Which protocol is chosen depends on the message size. For every MPI message the user program wants to send, a header respectively a control block is created. In this control block the size, the type and further information about the MPI message are included. All messages get a tag. The control block always gets the tag '0', the message gets a tag corresponding to the protocol. These tags are important for the probe and the reception of a message.

4.1.2.4.1 SHORT This protocol is used for messages smaller than 1.024 bytes. MPI header and MPI message are combined to one message. This message has the tag '0'.

4.1.2.4.2 EAGER This protocol is chosen for messages, which have a size between 1.024 bytes and 128.000 bytes. MPI header and MPI message are sent separately. The message gets a special tag: 1 + MPI_Source_rank. Hence, the receiver can recognize the source, which is important for handling unexpected messages. In both previous mentioned protocols all messages are sent immediately without waiting for a corresponding receive. This is a fast method but it could lead to a lack of memory at the receiver side.

4.1.2.4.3 RENDEZVOUS For that protocol the message has to be larger than 128.000 bytes. It is a handshake protocol. The header is sent with an acknowledge request (special message type). Then the message is not sent until the acknowledge arrives. So the receiver has the possibility to allocate enough memory space before the message arrives. The control messages have the tag '0', the data message the tag '1024 + number of pending messages'.

4.1.2.4.4 Message queuing The ADI from MPICH already provides a message queuing. It checks all control messages (messages with the tag '0'). If a control block indicates that an unexpected message will arrive, memory is allocated and the data message is added to a list of unexpected messages. In this context 'unexpected' means that no corresponding receive was posted by the user application. However a further message queuing inside the private ATOLL functions would be necessary because the order of control messages does not need to be the one of data messages. But ATOLL bypasses this problem with its pinned DMA space and the API functions 'Atoll_find_header' and 'Atoll_recv_desc'

4.1.2.5 MPICH/ATOLL Transfer modes

MPI provides two basic transfer methods, blocking and non-blocking. A blocking send, for example, blocks the user application until the message is sent. A non-blocking sent returns to the user program even if the sent is not finished. MPI does not guarantee the completion of this send until a blocking 'MPI_Wait' is started. However that MPI provides these two methods does not mean that every MPI device has to support both methods. With the definition of the two variables 'PI_NO_NSEND' and 'PI_NO_NRECV' in the file 'chdef.h' every device implementation can signal that only blocking versions are available. Then the ADI emulates the non-blocking functions with the help of the blocking ones. This is the way MPICH for ATOLL is implemented. We provide two blocking point-to-point transfer functions to the ADI and the non-blocking methods are emulated. The emulation proceeds as follows. Messages sent within the short and eager protocol are sent 'blocking' at once. The emulation presumes that messages with that size never block. Only long data messages within the rendezvous protocol are handled in another way. The control message is sent at once, but the data message is not sent respectively there is no attempt to send this message until the finishing 'MPI_Wait' is called.

4.1.2.6 MPICH/ATOLL Channel Device Functions

The following function were required to implement the channel interface.

- Atoll_Init()
- Atoll_End()
- Atoll_send()

- Atoll_recv()
- Atoll_probe()
- Atoll_Clock()

The `Atoll_Init()` function is responsible for the startup sequence as explained in 4.1.2.1. It starts the right number of processes using a `machinefile`, initialize a few important environment variables and establishes connections to all existing nodes. Using sockets as a communication transport, vital information can be exchanged.

The first process started by `mpirun` is the server process. It starts the remote processes using `rsh` or `ssh` and sets up a socket communication to each process. Every process initializes its local ATOLL hardware device from which it gets a unique identifier. The server collects all these ATOLL IDs and creates a table, which maps the MPI ranks to the ATOLL IDs. Eventually this table is returned to all clients. Now every MPI process knows the total number of nodes its rank and a reference to lookup MPI id and matching Atoll Id for communication.

The `Atoll_End()` function terminates the ATOLL communication with a call to `Atoll_finish()`. An extension to the existing ATOLL API was made by adding a tag to wait until all descriptors have been consumed.

Using `Atoll_Send()` a message containing MPI headers and possible payload is delivered.

If this message is larger than the ATOLL MTU defined through `ATOLL_MAX_SEND_LENGTH` the MPI message is fragmented. Within the MPI framework, the fragments get different header for the reassembly at the receiver side. This send mechanism relies on the low level send routines of the ATOLL network and can be considered to be nonblocking, as the message is copied into the DMA Send space and scheduled for transfer. The data delivery is handled by the hardware automatically and the MPI application can continue immediately since data has been buffered.

On the receiver side, the `Atoll_recv()` function is implemented as a wrapper of the original non blocking ATOLL receive function. It will block until a message arrived which provides a matching tag. The function also blocks until a divided message is completely reassembled. If unexpected messages arrive, it uses the out-of-order reception from the ATOLL API (see 4.1.2.4.4 for out-of-order reception). This can happen if messages from other sources arrive or if the messages from one source are unordered. After the reception of a MPI message two important global variables are set. The

`AtollLen` variable is set to the size of the received message and `AtollFrom` contains the sender Atoll id.

The `Atoll_probe()` function tests if a message with a matching tag arrived. This test could be implemented very efficiently because the ATOLL API provides non-blocking reception.

For time measurements, the `AClock()` function returns a timestamp with a resolution of a nano second.

4.2 The Parallel Virtual Machine using the ATOLL network interface

This section will describe the integration of the low level API of the ATOLL network into the Parallel Virtual Machine (PVM) framework. This work is similar to the work carried out in chapter 3. The integration of the ATOLL network interface allows applications which have enrolled into the PVM system to use the low level ATOLL API directly when establishing direct connections.

4.2.1 PVM Concepts

PVM has been a de facto standard for message passing. It allows a heterogeneous collection of machines seen as a single parallel virtual machine, abstracting from underlying hardware. This abstraction is also true for communication environments. For clusters of workstations the traditional PVM only supports TCP/IP communications. A detailed introduction into PVM can be found in section 1.2.2.1.

4.2.2 PVM ATOLL Implementation

The ATOLL network is designed to run on clusters of workstations very well. For this environment a scalable infrastructure is provided in which clusters can be enlarged by adding more nodes using an ATOLL interface card. A central instance will compute routing strings for each hostport. With each hostport having a unique ATOLL id, data paths from and to any combination of hostports will be available. Although ATOLL requires a setup of a connection pair for every two endpoints, there is no limitation in the number of connections. Moreover, the concept of establishing connections

is very dynamic and not depending on a process startup, like for example in the Quadrics Qs-Net network. For scalability, this mechanism even offers support for dynamic routing schemes. This could mean that the central instance which computes the routing information can take current statistics of data transfers into account and find an optimal routing path.

The idea for using the ATOLL network for point to point communication in ATOLL is to create a ATOLL connection using `Atoll_Connect()`. This connection will be established, when a send or receive operation to a target identifier is called. Then the PVM task will check if it has already enrolled in ATOLL, obtaining a unique ATOLL id. This Atoll id is then exchanged using standard communication through the daemons which service this protocol. Figure 4.2.2 depicts the control messages which exchange the necessary information. Using this protocol an efficient implementation can be achieved

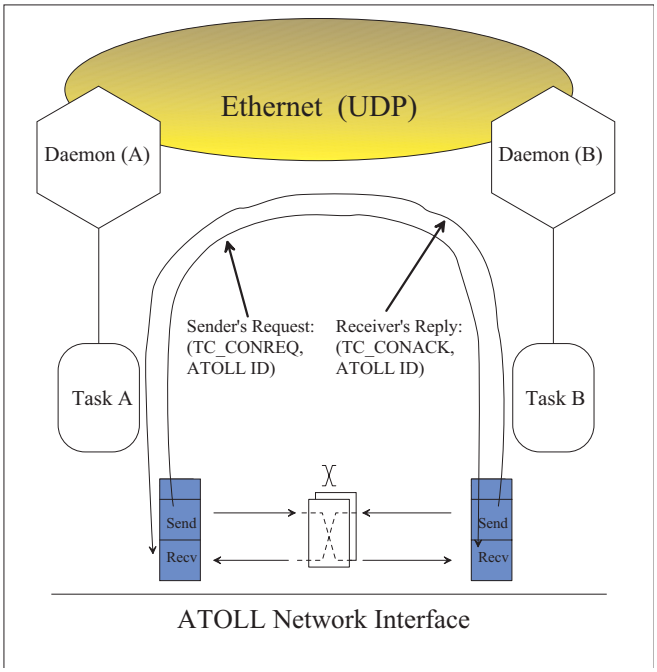

Figure 4.2: Establishing a direct point to point Connection using ATOLL endpoints

not relying on additional startup processes which would have to distribute the ATOLL id's in advance. With the described model, the ATOLL network is only requested by tasks which actually perform direct communication.

For registration into the PVM framework as described in 3, the following functions have been implemented.

- For implementing the send function, the `Atoll_Send ()` function can be used. The PVM system already includes support for message fragmentation, which has been adjusted to a maximum transfer unit in ATOLL. MTU in the ATOLL network have the effect that messages are unlikely to block and messages coming to switches can be addressed in a fair manner.
- Implementing the recv function for the PlugIn is also straight forward by using the `Atoll_Recv()` function. PVM typically reads first the header of on incoming message. This can be done by using the `Atoll_read_header()` function which has been filled with information about the message content.
- A select/poll mechanism was implemented by using the `Atoll_Probe()` function. Like any other function in the ATOLL API, this function is non blocking and therefore suited to implement a multi channel device.

Chapter 5

Design Issues for an Advanced ATOLL System Area Network

This section will describe an improvement of the existing Atomic Low Latency (ATOLL) design. It will introduce varying descriptors to achieve different protocols. These protocols can still be implemented directly in hardware without involving additional central processing units like for example in the Myrinet [9] or Quadrics network [49]. Thus, the overall goal of ATOLL which is to provide a network on a chip remains. The new extensions however, make ATOLL more flexible and efficient and will offer several enhancements for its usage.

5.1 Motivation

ATOLL already is an efficient implementation of a system area network. It is a very cost effective design and its 'network on a chip' design already includes typical external components such as a crossbar to build larger networks. Some design issues however have been neglected. The most important aspect is that of efficient host resource utilization. The following figures show the impact of low host utilization and can serve as a motivating point for enhancing the current ATOLL design. The overall goal for an enhancement is to keep ATOLL features, especially from the hardwares point of view. No rudimentary changes should be required.

Figure 5.1 depicts the performance enhancements for file transfers when improving traditional protocols. This improvement can be a totally different approach when using a user level file system like Direct Access File System

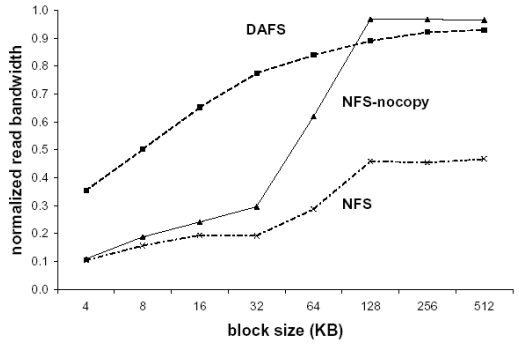

Figure 5.1: Bandwidth Improvements using Remote Direct Memory Access [125].

(DAFS) or a modification of existing protocols to use remote direct memory accesses (RDMA). As a result DAFS already gains high bandwidth for short block sizes. There is a threshold value for zero copy network file system (NFS) implementations after which the traditional NFS scheme is outperformed significantly. Insufficient memory bandwidth has been pointed out to be the cause of the weak performance of NFS [125].

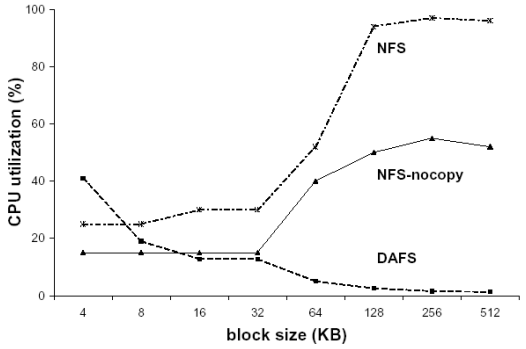

Figure 5.2: Host Utilization Improvements using Remote Direct Memory Access [125]. Efficient protocols lower host utilization significantly. User level protocols deliver the best results.

Moreover, the host utilization can be reduced significantly if the network interface is capable of RDMA. Both, user level direct file access as well as zero copy NFS implementations lower host utilization significantly. The very

memory intensive NFS protocol require multiple data copies throughout the protocol stack.

5.1.1 Limitations in ATOLL1

The overview on ATOLL1 given in section 2.3.4 described in which way messages are transferred. Currently, ATOLL1 uses DMA transactions on contiguous send and receive buffers for midsize or large messages. When sending a message, data will be copied by the host processor into the SEND DMA region which is implemented as a round robin buffer. The ATOLL hardware will be triggered by incrementing the write pointer of the descriptor region. The hardware will then read the descriptor and start a DMA engine which will use an offset in the SEND DMA region to inject messages directly into the network. On the receiving side, the data will be posted into the RECV DMA region and a descriptor will be assembled describing the new message. The host processor will then be used for copying the data to its final destination. For very small messages, the host processor can inject messages using Programmed I/O (PIO).

This communication mechanism using DMA engines which take data out of fixed allocated buffers, is known as a 1 copy or buffering strategy. It has proven to be very efficient for smaller messages. These buffers can be allocated dynamically during runtime [9] and can be adaptable in their size. ATOLL1 manages its buffers itself. When a system starts up, physical memory can be excluded from the OS. This amount of memory will be used to be split up for several host interfaces. Since the regions are round robin buffers, the messages can only be received in chronological order, assuming an efficient ATOLL API implementation. Otherwise the regions can be fragmented and interrupt handler would be required to rewrite descriptors after compacting the memory regions. This however should not have a high impact on higher protocols since they typically provide handling for unexpected messages. For larger messages however, a zero copy strategy which implements a direct application buffer to application buffer message transfer, would be more efficient. A zero copy strategy would leave the CPU out of the message transfer. Therefore, the cache which will be dirty when copying large amounts of data, would not be affected. But also, the process could perform overlapping computation.

5.1.2 Protocol off-loading

Demand for networking bandwidth and increases in network speeds are growing faster than the processing power and memory bandwidth of the compute nodes that ultimately must process the networking traffic. This has become more important since industry starts migrating to a 10 Gigabit Ethernet infrastructure. Protocol off loading as well as Remote Direct Memory Access (RDMA) in combination will be capable of providing required efficiency.

Several existing server application exist which use the socket interface when communicating to their clients. The number of transactions is therefore closely depending on how much time the processor has to spent on serving a single request. While network speeds have increased, Ethernet does not have experienced higher efficiency [95]. For Ethernet protocols, a TCP offload engine (TOE) has been developed as a specialized (intelligent) network adapter that moves much of the TCP/IP protocol processing overhead from the host CPU/OS to the network adapter. However, while TOE's can reduce much of the TCP/IP protocol processing burden from the main CPU, it doesn't directly support zero copy of incoming data streams. Contrary, RDMA directly supports a zero copy model for incoming data to minimize the demands on host memory bandwidth associated with high-speed networking.

5.2 Zero Copy Mechanism in General

Recent research tries to avoid unnecessary data copies which results in a so called zero copy mechanism, where data is directly fetched from its position in application memory and directly deposited in remote application memory. Using this method, it is expected to decrease latency and increase bandwidth for data transfer. Basically, if PIO is available, this communication mode can be used for zero copy. When sending, data is directly injected by the CPU into the network. On the receiving side, the message can again be delivered directly with PIO. The disadvantage is that the processor will be involved during the entire transaction and can not be used for computation during that time. To enable the DMA engine to perform this task, a virtual-to-physical address translation must be implemented, which increases hardware complexity significantly. Sharing the page tables between the OS and the device is complex and time consuming too. The TLB handling is usually performed by the OS. Pages for translation have to be pinned down, and virtual addresses now represent physical ones. The TLB can be placed and managed at NI memory, the host memory, or both. Using this method,

zero-copy can be achieved via remote memory writes using the information provided with the TLB.

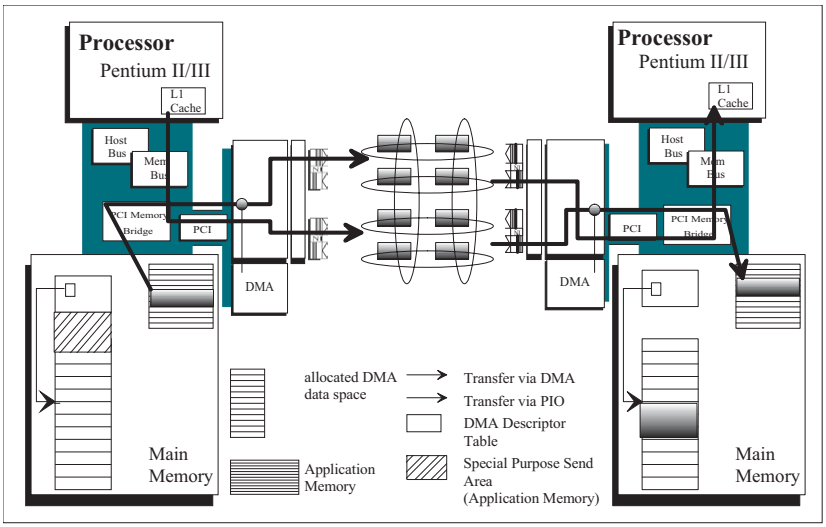

Figure 5.3: Upview of Direct Transfers. A message can be directly deposited by using either PIO or DMA data transfer mechanisms. PIO involves the CPU during data transfer, while DMA engines require virtual to physical address translations. A notification mechanism which signals the end of the data transfer is needed as well for DMA transactions. Host utilization however can be reduced.

Without a virtual to physical address translation, send and receive operations carry the physical address of the destination buffer and the DMA engine copies the (contiguous) data directly to the destination address. To add additional protection, a combination of a virtual address and a unique identifier such as the process id is used. Typically, a rendezvous model is needed before such operation can take place, since the location at the receiver side is not known a priori. A requirement for the NIC is to access dynamically pinned down data. This DMA method will show better efficiency, if the pages containing the data to be transferred is locked down once and the segment can be re-used. Otherwise expensive lock and unlock sequences will lower performance making a trap into the system. Typically, a caching strategy implemented as part of the low level API will hold data structures of already registered memory pages. Pages are de-registered if the maximum number of pages are reached or the application terminates. Another problem coming along with zero copies is that of message flow control. It is not obvious when

a message has been transferred and the space can be used again. On the other hand, support for remote DMA eases the implementations of one sided communication.

5.2.1 Security

A direct memory access through a network device can weaken a host system if the communication protocols and implementations are not adding extra protection. On a modern multi tasking operating system, each process typically has its own address space and references to memory pages not belonging to the same process will result in a segmentation fault (SEGV). When doing direct data transfers, the network interface hardware will copy data from an application buffer into the network and deposit it directly on the receiver side. But also one sided communication like the *get()* method will fetch data using direct transfers. When doing so, the host to NIC DMA engine is only using physical addresses and no further checks by the processor using memory tables can be done. With current hardware which is using the PCI bus, the synchronization with data structures in host memory which describe the virtual memory layout is rather difficult and requires a tremendous amount of work. This often leads to complicated extension to the kernel and modularity is reduced. The Qs-Net from Quadrics is such a NIC which is only available for the proprietary True64 operating system. Thus, standard networks like Myrinet exchange the virtual addresses prior to sending the actual data.

5.2.2 Address Translations

5.2.2.1 The Memory Management Unit (MMU)

During runtime, an application holds virtual addresses. These addresses can be the same for multiple applications. Together with the process id, the virtual address is translated uniquely by the memory management unit (MMU) to a physical address in main memory. While an application operates on virtual addresses, and a page fault would be handled by the operating system, it is required for communication that the data is locked, thus resides consistently in physical memory. This is required since a NIC has no MMU functionality to handle virtual addresses but also would not be able to page memory on demand.

5.2.2.2 Memory Registration

With current architectures which are the distinguished research platform for this thesis, network devices and system busses are physically and logically separated. For direct transfers this means that it has to be made sure in advance that message transactions are secure. This means that when exchanging data, the source and destination have to been accessible. One might distinguish between sending and receiving messages. Typically a message being send has been touched before recently and the data will be available. However when receiving, the host system may have placed data into different physical pages. Fixing the address translation from virtual to physical addresses are the main intention of memory registration. Since the network device is not able to perform a lookup itself, this translation has to be made available to the network device in a proprietary form. As an alternative, the network device would need access to the host kernel memory structures obtaining a pointer to the MMU structure of the host system, the operating system respectively. These MMU data structures however vary from time to time. Since a network device should not be bound to a specific operating system it would have to implement its own MMU in a portable manner. The ATOLL network approach is to be implemented as a 'Network on a Chip', with a very slim design using standard components. Therefore, the implementation of its own MMU in an 'RDMA enabled ATOLL' should not involve to much overhead. Section 5.5.2.3 will provide further detail on how the design solves this issue.

5.2.3 Message Transfers with Zero Copy

The following figure depicts a message send and receive scenario using a buffering method and another one bypassing the operating system as well as intermediate message copies locally.

Typically, the message is copied from application memory into an intermediate buffer, again in application memory (for example to pack multiple data together and to be able to continue to compute by not modifying message data). A send operation then moves the data to the kernel which will inject the message into the network using a protocol which observes flow control. This mechanism not only involves additional copies but also slows down the application by a trap into the system. User level communication prevents the latter by having direct access to the NIC. By pointing the NIC to the message in user memory, copies to a reserved data space can be avoided and

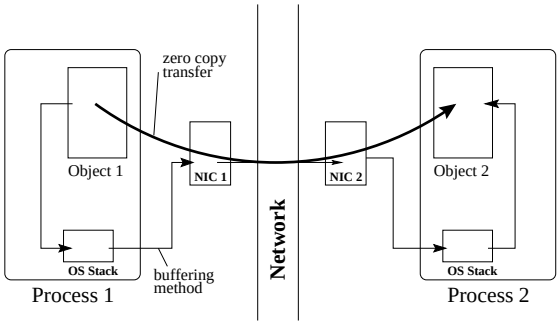

Figure 5.4: Avoiding Message Data Copies

higher performance can be achieved. The performance is depending on message size since additional mechanisms are involved to establish this behavior (page locking). As described in section 5.2.2 one important requirement for zero copy is to dynamically load and store message data. Dynamically in this context means that it is not required to have a fixed space for message transfer.

5.2.4 Related Work

High speed networks have been popular since several years. Especially the concept of user level communication has been driving the system area network research. With direct control over the device the concept of zero copy data transfers has been made possible to be directly supported by hardware. In the following related work on zero copy transfers using system area networks will be presented.

5.2.4.1 Shrimp

The Scalable High-performance Really Inexpensive Multi-Processor (Shrimp) project was targeted at providing high performance servers using commodity PCs using an commodity operating system. It consisted out of several areas. The communication part itself was working on user level and protected communication and efficient message-passing, In this research their VMMC [62] protocol was enabling zero copy transfers.

5.2.4.2 SCore

Another project having gained popularity is the SCore project [98] from the Real World Computing Partnership. The original goal was to develop a massively parallel computer for which every component should be implemented by themselves. The SCore Cluster System Software in the end provides a high-performance parallel programming environment for workstation and PC clusters. The main features SCore provides are a Single System Image View, multiple network support, and multiple programming paradigms. This way SCore users are not aware whether or not a system is a cluster of single/multi-processor computers or a cluster of clusters. The PM II high performance communication library is a dedicated communication library for cluster computing using many types of networks. It allows a program to communicate on different types of networks such as Myrinet and Ethernet. This driver comes with true zero copy functionality. Unlike other cluster software, SCore not only supports the message passing paradigm, but also supports the shared memory parallel programming paradigm and the multi-threaded parallel programming paradigm. Finally, fault tolerance has been increased by providing preemptive checkpoint at driver level. This way, the checkpointing is totally transparent to the application.

5.3 Zero Copy Implementation Alternatives

Before the extensions on an advanced ATOLL SAN will be presented, a short overview on available alternatives will be given. This overview will also provide an analysis justifying the method which was finally implemented. Basically two choices exist:

- Page Flipping and Copy on Write (COW)
- Direct Data Placement

Using page flipping and copy on write, sending and receiving must be differentiated. For this method, the interaction of the operating system is required. When sending, the operating system will put a COW mapping on each page the application writes to a socket. The data the user program writes must be page sized and start on a page boundary in order for it to be run through the zero copy send code. If the application does not write to the page before it has been sent out on the wire, the page will not be copied. Otherwise, the COW on write mechanism will detect a reuse of the

buffer and will copy the pages into a buffer pool. This way, a send side zero copy using COW will only better if the application does not immediately reuse the buffer. When receiving, the NIC driver receives data into buffers allocated from a private pool. If the application reads the data, the kernel page is substituted for the applications page and the applications page is recycled. This concept is otherwise known as 'page flipping'. This page flip can only occur if both, the application buffer as well as the kernel buffer are page aligned, otherwise it must be copied. Another requirement for zero copy receive is that the chunks of data passed to the network have to be at least page sized, and be aligned on page boundaries. This requires support from the NIC to have a MTU of the page size. Thus, this concept comes with several disadvantages.

When using direct data placement, the NIC uses headers (descriptors) which describe the origin or destination of a message. It therefore steers the payload directly into application buffers. A origin or destination can be described by physical or virtual addresses and this method is independent of the MTU, the page size of buffer alignment. It can also be implemented in a low level API and will be therefore compatible with existing applications. This method however typically requires a short rendezvous protocol which exchanges the source or destination origin to set up a descriptor. Another aspect is that of MMU functionality. If virtual addresses are used in combination with a process id, a translation to a physical address has to be made. A NIC having a MMU is currently only implemented in the Qs-Net [49], which is very operating system dependent (True64 in this case). A port to Linux is marked as work in progress. An example of a implemented direct data placement strategy are TCP offload Engines (TOE) which steer the payload, if receive buffers have been pre-posted.

The method of using page flipping and copy on write involves the operating system and NIC to some degree. A page remap will also require the shootdown of the TLB for SMP's. The latter method can be implemented in a portable fashion and is therefore the better approach.

5.4 RDMA Using Message Handlers

The old approach, moving data through I/O-channel or network-style paths, requires assembling an appropriate communication packet in software, pointing the interface hardware at it, and initiating the I/O operation, usually by calling a kernel subroutine. When the data arrives at the destination, hardware stores them in a memory buffer and alerts the processor with an

interrupt. Software then moves the data to a temporary user buffer before it is finally copied to its destination. Typically this process results in latencies that are tens to thousands of times higher than user level communication. These latencies are the main limitation on the performance of Clusters or Networks of Workstations.

5.4.1 Software and Hardware Message Handlers

In ATOLL message handlers exist that continuously receive incoming messages. The implementation of such a message handler is an autonomous DMA engine which spools incoming data automatically into the provided location in user space. For independency, a message handler exist for each host port.

In this context the steps in receiving a message are depicted in the following figure. When a message arrives the message it is first copied to the

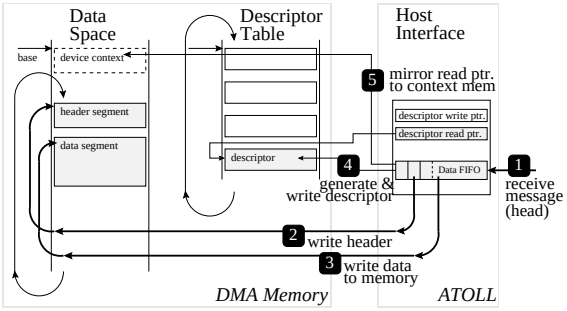

Figure 5.5: Steps Receiving a Message

provided DMA receive buffer using the current write pointer. This write pointer is then increased by the message length and a receive descriptor is assembled and stored at the current descriptor write pointer. It is obvious, that the current write pointer has to be modified, pointing to the application memory to which the data has to be transfered.

5.4.2 A protocol for Zero Copy / One Sided Communication

To allow for zero copy message transfers, the following requirements are needed to perform this task. First is an accessible and known location in

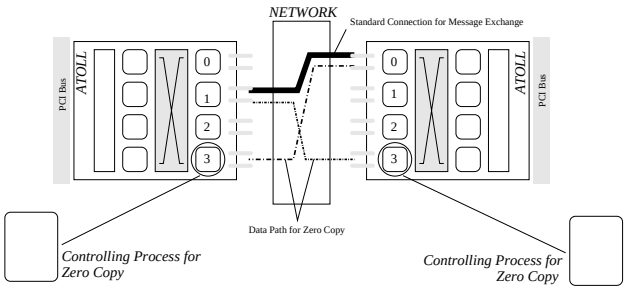

Figure 5.6: Zero Copy Protocol for ATOLL

application memory to store or fetch the data. Finally the NIC needs to be programmed to access the data at the desired location. While the first requirement is mainly a software problem in which application memory is typically pinned down during the time for transaction and a virtual to physical translation takes place, the second requirement needs modification of the hardware settings. Since in ATOLL parameters such as routing table and other DMA regions are mirrored, the NIC's behavior can be modified during runtime.

For this purpose the modification of the write pointer will bring the DMA engine to spool the message to its final destination, bypassing the ring buffer. Unfortunately, the hardware of the first version of ATOLL will not be able to modify the write pointer by itself, if this new location should be encapsulated at the beginning of a message for example. However, by reserving one hostport under the supervision of a controlling process, true zero copy can be enabled. Figure 5.6 depicts two communication partners and their standard routing using hostinterface 0 and 1. Choosing hostport 3 to handle zero copy, hostport 0 and 1 will then send messages to hostports 3 which will deposit the messages to its final destination under the supervision of a controlling process.

The following protocol establishes a zero copy protocol for multiple tasks, a send/recv scenario Task A to Task B is assumed.

- Task A sends a PIO message, containing final destination, to the controlling process/reserved hostport
- The controlling process modifies the DMA data region base information of the DMA engine in the reserved hostport
- The controlling process acknowledges with a PIO message

- The message can be send via DMA using the reserved hostport
- The message has been delivered once the hardware assembles a descriptor for the new message

This way it is made sure that an incoming zero copy message has been pointed to its final destination, but also further messages are held back until an acknowledgment is being returned from the controlling process. Finally, the data placement can be monitored by checking on the message descriptor. This can be used as a DMA-ready checking which is not available using other NIC's.

5.5 ATOLL RDMA Using Protocol Extensions in Soft- and Hardware

In this section, the design of an enhanced ATOLL and its implementation using the current ATOLL software environment will be presented. This approach of enabling ATOLL for RDMA transactions is based on message handlers implemented in hardware which are triggered upon a special command byte which will be shipped with the message.

5.5.1 The Actual ATOLL Environment

The ATOLL Environment can be distinguished by user level and kernel level components. Both, together with the ATOLL NIC or a simulator which mimics I/O behavior of the ATOLL device, form a ATOLL environment.

5.5.1.1 User Level Components

An application has to be linked with the ATOLL library, which will enable access to the ATOLL device. In order to use ATOLL, the `atoll_init ()` function must be the first function to be called. It will return a handle to one of ATOLL's hostports. Using send and receive functions like `atoll_send ()` or `atoll_recv()`, messages can be exchanged using the ATOLL device. In order to do this, the ATOLL library will abstract from the underlying device and only provide higher level communication routines. Internally, the ATOLL library will modify descriptors associated with messages, will update their write and read pointers and will also buffer messages into the DMA Send

space. For this functionality, the ATOLL library maps physically contiguous data into the applications memory space. This chunk has been initialized by the ATOLL driver, which is a kernel level component. This concept is actually different than that of typical network cards. Usually a system trap is required for sending messages since the device is shared. With ATOLL, each process owns a dedicated hostport and therefore ATOLL allows direct access to it. This access also does not involve additional overhead. The ATOLL library can modify memory directly.

ATOLL will not work before this driver is loaded. Figure 5.7 gives an overview on how these components have a relation. Since ATOLL requires physically contiguous chunks, they have to be allocated somehow. One way is to apply patches to an operating system in order to gain access to large chunks of contiguous data. This is normally not the case. Linux as primary platform for ATOLL and the cluster market offers to specify which memory is available. If less memory is specified as it is physically available, then the remaining amount of memory can be dedicated to the ATOLL device.

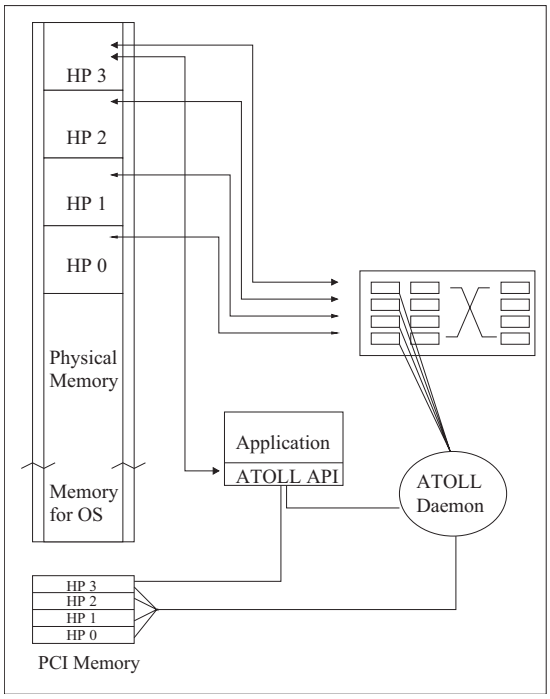

Figure 5.7: The ATOLL device interfacing with User Level- and Kernel Level Components

5.5.1.2 Kernel Level Components

The ATOLL Device Driver is a kernel component which enables the ATOLL network. It is implemented as a module and will initialize the ATOLL network upon insertion into the operating system. Major steps during the initialization are the mapping of reserved memory for ATOLL. This chunk of data is then segmented to reflect all available hostports. With user space memory mapping, Linux allows direct access to the device. There exists however two functions, which will require the interaction with the kernel components: the `atoll_connect()` and `atoll_disconnect()` functions. When calling these functions, the routing information from *source* to `destination` is queried from a central instance and stored in a reserved space. A descriptor will point to the offset in this region to let the NIC fetch the information. This is however no real performance loss, since connections are established once only. The interaction with the kernel is quite obvious. The ATOLL daemons which are connected to a master daemon will receive the optimal routing information from the central instance.

5.5.1.3 Kernel Process Mimicking ATOLL Hardware

In order to test the user and kernel level components, a simple simulator is available which mimics the final ATOLL hardware. This simulator will map the chunks of data which resemble the hostports. It will then poll on the descriptor space to see if the ATOLL library called ATOLL's send or receive functions. The simulator will then transfer the according to the routing information. This can be a hostport on the same node or a remote node. Locally, a memcpy operation will be sufficient to transfer the data, otherwise socket operations using a standard network are used.

5.5.2 Protocol and Descriptor Enhancements Autonomous RDMA Transactions

Before a detailed description of the adapted protocols and descriptor will be presented, a short introduction about the final implementation will given.

5.5.2.1 Address Translation using the Locked Memory Manager

The locked memory manager is a software implementation available as a Linux module [96]. As stated earlier, zero copy transfers using the PCI

interface and the Linux operating system require the locking of memory pages to prevent them from being paged out. Locking of memory requires kernel level functionality and will end up in the mlock() function being called. Therefore, the locking mechanism which actually only sets a bit in the MMU structures is an easy task to perform. The Locked Memory Manager (LMM) [96] however provided other advanced features which for example dealt with the overlapping of segments and their multiple registration. The LMM in its current version uses a new mechanism called kiobuf (Kernel I/O Buffers), which was introduced in version 2.4.x of the Linux kernel version. Kiobufs were originally implemented in conjunction with Raw I/O, which allows data transfers between disk and user memory without intermediate copies by the kernel. During data transfers the associated memory pages must be locked. A kiobuf can be mapped to a part of the user address space. After that the physical page addresses can be read directly from the kiobuf, and the page tables don't have to be touched. The Linux 2.4.x version introduced separate functions to lock and unlock an already mapped kiobuf.

5.5.2.2 RDMA Protocol Description

Before the final protocol implementation will be described a short analysis about the implications for ATOLL will be given.

The current ATOLL descriptor consists of 64 bytes, one cache line respectively. For an enhanced ATOLL, which should also be implemented as a FSM, this descriptor should not be increased in its size. However, depending on a special tag it should be possible to provide a different layout of the descriptor. Currently, the descriptor contains an offset to the message in the DMA Send space. In combination with a length, the ATOLL Device will inject data into the network. This mechanism has been intensively described in section 2.3.4. In the following additional protocols which extend the original ATOLL functionality will be presented.

An enhanced ATOLL device should be able to allow one way communication. That is a direct send will transfer data directly from application to application buffer without intermediate copying. Thus for the implementation for an enhanced ATOLL device, the offset which has been used to point to the actual payload can not be used since it is not available within the limits of the Hostports. Therefore an absolute addressing scheme is required. There are two choices for absolute addressing. Both, physical as well as virtual addresses in combination with a unique identifier can be used. Using virtual addresses adds some security since a translation from virtual addresses to physical addresses has to be made. This usually requires a TLB

which will report errors if a lookup of the virtual address can not be found. The address translation and lookup mechanism will be explained in further detail in section 5.5.2.3.

Finally, the direct send will also be delivered directly to its final destination on the destination node. Therefore, an absolute addressing for the remote memory space must be required as well. As a consequence for an advanced ATOLL, the descriptor will use the given Command Byte in order to distinguish between different protocols. The command byte will then be able to specify a normal message (N), which uses the buffering mechanism, a put message (P) which will result in a direct store as described above or a get operation (G) which will fetch data from a remote host.

The following additional ATOLL API functions allow zero-copy data transfers.

```
/* Registration of len bytes starting from va */
int Atoll_register_memory (char *va, size_t len);

/* De Registration of len bytes starting from va */
int Atoll_deregister_memory (char *va, size_t len);

/* Direct Store of len bytes from va_self at dest
                                    address va_dest */
int Atoll_put(atoll_id self, atoll_id dest,
              void* va_self, void* va_dest, size_t len);

/* Direct Remote Load of len bytes from va_src at
                              node src at va_self  */
int Atoll_get(atoll_id self, atoll_id src,
              void* va_self, void* va_src, size_t len,
              uint32 src_rtoff, uint32 src_rtlen );
```

With the RDMA instances described above, the interaction between user- and kernel level components becomes slightly more complex. Figure 5.8 depicts its relation.

5.5.2.3 RDMA Address Translation, Lookup and Security in ATOLL

Given the logical separation of operating system data structures and current network devices, their synchronization is one crucial task which needs to accomplished to establish zero copy transfers.

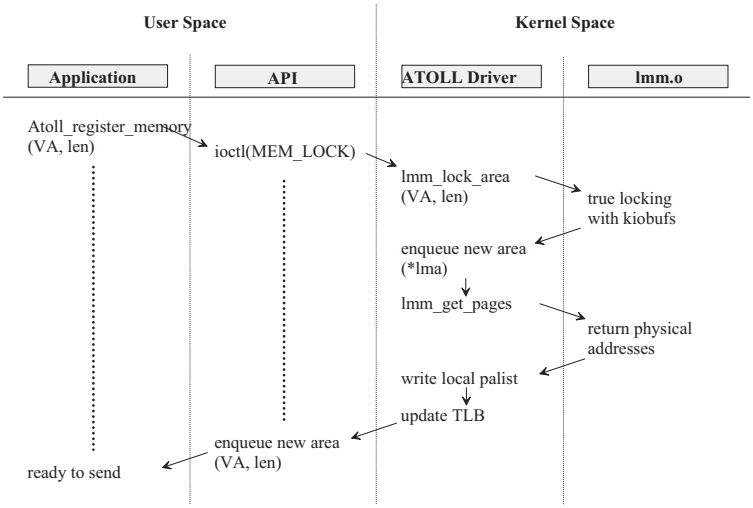

Figure 5.8: RDMA Components Interaction

As mentioned before, the access of memory regions by the network device can not be observed by the MMU. The network device has to operate on physical addresses. To guarantee security, the memory pages for direct transfers have to be locked before. During this operation, the ATOLL library as well as the network interface will be able to reference a virtual to physical address translation. The locking is also considered being a memory registration, however a communication phase is not imitated yet, but is a requirement. When sending a message directly using the `put (P)` mechanism, first the ATOLL library will check whether the virtual address va_src has been registered. This can be done by querying the internal LMM structures. It will deny to initiate a `put (P)`, if a lookup of va_src can not be found. Upon success however, the ATOLL library will assemble a descriptor (specifying a `P` Byte) and the ATOLL device will be triggered by incrementing the write pointer. The ATOLL hardware will first read the Command Byte and detect a direct message. It will then use a TLB mechanism to lookup the virtual address to determine a pair of physical addresses and their length. Different from the current ATOLL implementation, multiple DMA transactions can be required to deliver a message. This is because for larger messages, the data to be fetched may not be contiguous. A virtual address is therefore stored as a set of physical addresses VA = $\{(Phys_(VA)(0), \text{len}(0)), ..., (Phys_(VA)(\text{n-}1), \text{len}(\text{n-}1))\}$ having n physical entries. For this set len(1) .. len (n-2) have value of the page size of the used system while len(0) and len (n-1) have value 1 .. page size. Figure 5.9 shows how a potential setup could look like.

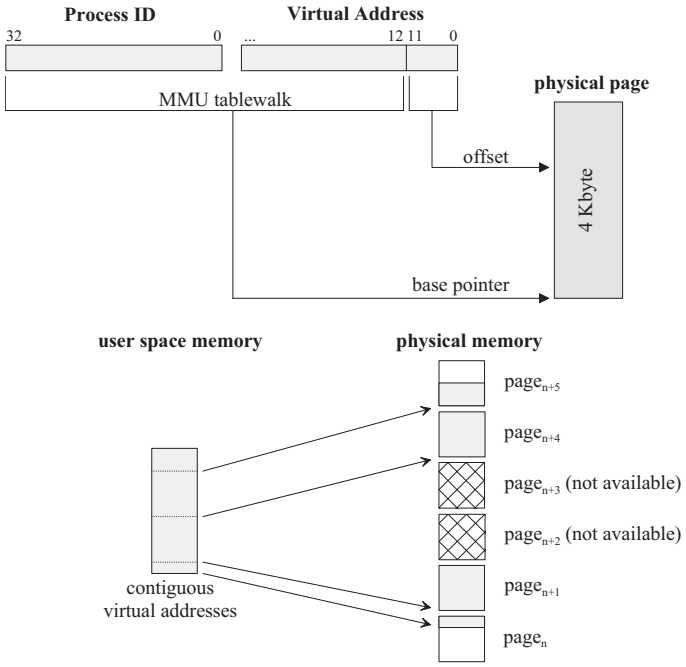

Figure 5.9: Operating System Memory Layout

The ATOLL device will inject the messages as one contiguous data stream into the network. Figure 5.10 depicts the layout of the message header.

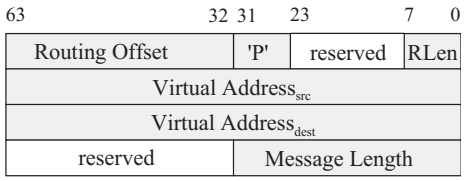

Figure 5.10: The Descriptor Layout for the Pet Protocol

Eventually, the message will reach the hostport of the remote hostport. The ATOLL device has been extended to check the transfered command byte. In this case it will detect a put message and will use the following 64 bits in order to determine the virtual address (va_dest). Like the initiating ATOLL device it will now lookup and match the given virtual address and translate it into a set of physical addresses and their corresponding length. Since the ATOLL message is available as one contiguous stream, there are no problems

for the receiver to dispatch the message according to its memory layout and the mapping of virtual to physical addresses. A much more complex and not suitable method would be to inject multiple messages at the sender side having different message sizes than the physically contiguous memory chunks at the receiver side.

When remote data is requested using the `get (G)` mechanism, a descriptor will be assembled which contains the remote virtual address, the local virtual address and the length of the message. The transaction which will be performed is to retrieve the remote data into the local buffers. This mechanism should not involve any remote host system actions. The reason for this is quite obvious. For a one way communication using a *get* mechanism there is only one communicating instance. The remote host does not know about the transaction, hence one way communication, and therefore will not schedule resource for its handling. Otherwise a resource, e.g a polling thread, would have to check continously for incoming requests. This would however waste resources. In the RDMA ATOLL design, the initiating host will assemble a message which contains the G command byte. On the destination side, this command will be extracted and the ATOLL device will enter its routine for handling messages which request the *get* mechanism. For this, the remote ATOLL device which receives this request not only has to receive the message, but it also has to respond to it. To put this into the given FSM design, the ATOLL device will assemble a descriptor containing the *put* (P) command. This mechanism fits the current ATOLL implementation well, since it already assembles a descriptor autonomously itself, when a message has been received and has been stored in the provided RECV data space. For sending a message, the ATOLL device will take all necessary information out of the corresponding descriptor. It will therefore require an offset to the routing bytes for the destination ATOLL id, the virtual address where this message has to be fetched from, the remote destination virtual address where this message will be stored and the message length.

Thus, the ATOLL device, when responding to a G command, will take the next 2 64bit and store the first 64 bit as va_src, the next 64 bit will be va_dest. Consequently it will continue to fill the descriptor with the provided offset and the message length. Figure 5.11 depicts the layout of this incoming message.

The delivery of this message can now be initiated by incrementing the descriptors write pointer. In this case the ATOLL device will detect a new descriptor which specifies the *P* command byte and will deposit this message into the remote address space. It may be noted, that the actual message retrieval is based on the *put* mechanism explained before. Therefore, it will

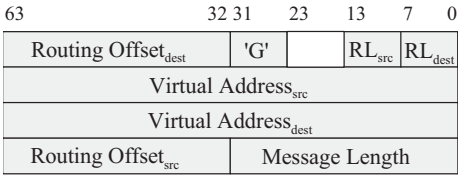

Figure 5.11: The Descriptor Layout for the Get Protocol

also check the virtual addresses and perform the same address translations and lookups. Finally, this completes the original *get* function.

In terms of security, RDMA allows direct memory accesses. With one way communication functions, this is a potential security whole if data can be simply taken out of a remote address space using the `get` mechanism for example. In order to address this, the RDMA enabled ATOLL will have additional functionality which will be available in User- as well as Kernel Level Components. Therefore the RDMA enabled ATOLL will add a device centric register to allow for RDMA transactions. It will also add hostport centric registers to let a hostport grant or deny access for RDMA. For this the kernel level component will overrule any request made by the user level component. That is, ATOLL can be enabled for RDMA when inserting the ATOLL driver. It also can be restricted from RDMA features. For an application, it will use the

```
int atoll_allow_rdma (atoll_id id, int how);
```

command to grant or deny RDMA access. The initial setting should be to not allow RDMA transactions. The implication for the transfers will be that the ATOLL device will have to check whether

- If the ATOLL device has been initialized with RDMA enabled
- If the corresponding hostport has allowed for RDMA transactions

This will not have a performance impact but will enable security. If one of the above disallows RDMA transactions, then the ATOLL device will simply discard the message. In the following an efficient design for a RDMA ATOLL address translation will be presented. The implementation was done in [126]. The following description reflect the result of this work. Since the address translation must be visible for the low level API as well as the autonomous

NIC, two extensions were added to the ATOLL environment and have been replicated for each hostport.

First is a small onboard SRAM which contains a Translation Lookaside Buffer (TLB). For efficiency, this TLB will be located on the real NI hardware. Second is a memory segment to hold the lists of virtual to physical addresses, which point to the message buffers. This area is called PALIST within the ATOLL implementation. It is worth noting that the host MMU requires not only the virtual address, but also the process identifier for a correct address lookup. This design relies on the fact that each hostport can only be accessed by exactly one process. Therefore an outgoing or incoming address is marked by a hostport and therefore the lookup is limited to a hostport.

Since this concept is independent from any OS, the data structure for storing the references can be chosen to fit an efficient lookup and implementation in hardware.

How virtual addresses are resolved as described in [96]. For each message buffer only the virtual start address is written to the TLB. All physical addresses of all pages (its individual physical address list), which are allocated for the message buffer, are stored in the PALIST. In order to get the individual physical addresses of a memory area, the LMM function lmm_get_pages is used. The TLB entry for each message includes the offset to the PALIST base pointer and the pointer to the individual address list, respectively.

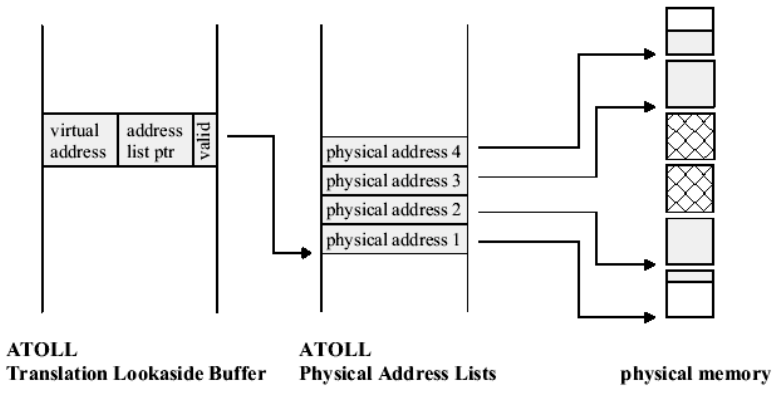

Figure 5.12: PALIST structures referencing a virtual address

As mentioned above, the implementation is designed in a way, which should make as least as possible changes necessary to the existing ATOLL

environment. In order to support this strategy the same descriptor mechanism, which has already been introduced for the buffered DMA transfer, is used for the zero copy functions. Even the size of the descriptors remains unchanged. These descriptors signal the NI (and the simulation, respectively), that a new zero copy message has to be sent. The only difference to the standard, buffered send API functions is, that the message does not need to be copied to the DMA space any more. In addition, the contents of the new descriptors have changed. They have to signal, that a zerocopy transfer is used and they have to provide the virtual addresses. Then the simulation has to react accordingly, which makes additional routines for the simulation necessary. Figure 2.7 shows the standard descriptor (for buffered DMA transfers). The new put and get descriptors are depicted in figures 5.10 and 5.11, respectively. The major differences are that the standard descriptor includes offsets to the pinned DMA space, the zero copy descriptors include the virtual addresses to the send and receive buffers. The length of zero copy messages is stored in the tag. In order to distinguish the different descriptors the command byte is *N* for normal descriptors, *P* for put descriptors and *G* for get descriptors. Another difference is, that the get descriptor needs two more values: the routing length and the routing offset of the source node. The problem emerges through the get - transfer. This transfer is started by the receiver, which sends a message only containing a get - descriptor. This message triggers the NI of the sender, which copies the source buffer directly from user space and sends it back to the receiver via a new created message. However, the CPU of the sender should not be interrupted at all. Hence, the NI needs all necessary informations for the new message. The virtual address, the translation to physical addresses and, moreover, it needs the routing to the receiver. This information is made available by the routing offset and the routing length of the source node. In addition, the previously shown mechanism means, that the receiver has to know the routing offset and length of the sender. It is not supported by the current ATOLL API, as each node only has to know one way of the routing. Two solutions are reasonable. Firstly, the routing informations could be exchanged during a connect (possibly a parameter signals whether or not zero copy transfers are supported). Secondly (which describes the current implemented), the rendezvous control messages include the additional informations. Furthermore, the simulation is changed in order to support the new zero copy transfer mechanisms. The simulation consists of a new switch statement, which distinguishes the descriptors. If a put descriptor is found, the virtual address is read. Afterwards the TLB is searched through for the virtual address. If it is not found, the simulation returns an error. The hardware would have to produce an interrupt in such a case. Nevertheless, that case should never

occur. If the correct TLB entry is found, the offset to the address list is read. Afterwards one physical address after another is read from the PALIST, the length for a copy transaction to the temporary buffer is calculated and the copy is performed. Then (as within the standard, buffered transfer) a receive function of the correct hostport is called. This receive function also distinguishes between different descriptors and, if a put descriptor is found, it also reads the virtual address, translates it and copies the contents of the temporary to the physical addresses. The difference to the standard receive is, that no receive descriptor is created and no message is copied to the receive DMA space.

Another aspect is that of a race condition if the hardware assembles a descriptor to trigger itself for sending out a *put* message, as a response to an incoming *get* request. While the data structures to create a descriptor are available, it could interfere with the ATOLL library which is also creating a descriptor itself for sending out a message. In this situation, a race condition must be prevented. The current ATOLL implementation offers a semaphore which however comes with a cost penalty. An alternative would be to provide an device descriptor entry residing on the NIC which would store the information. In this case, the ATOLL device would be triggered by the write descriptor and the modification of a local descriptor.

5.6 Conclusion

An extension to the existing ATOLL network has been presented, which allows for RDMA operations. That is, the ATOLL network is able to transfer data without the involvement of the host CPU. This extension requires only small modifications to the current ATOLL which was one of the design choices. Furthermore, a detailed description on how this RDMA enabled ATOLL can be setup for modern operating systems has been presented. Costly hardware components like a programmable controller or additional onboard memory can be avoided by the presented solution.

Chapter 6

An Efficient Socket Interface Middleware Layer for System Area Networks

The following chapter describes a new and innovative middleware layer called *Sockets Direct* which will effectively map existing distributed applications using the socket interface to a System Area Network (SAN). One of the most important features for such a middleware layer is a SAN's capability to provide reliable message delivery without additional protocol overhead. Among others, these features are the basis to overcome overburdened protocol stacks to let applications fully exploit the available raw performance from high speed networks.

Modern high speed networks which are considered being a SAN offer automatic error correction and detection. Thus, the low level API of a SAN can be implemented very efficiently assuming only a severe error, like a broken or dead cable can be the only reason for data corruption. An overview on its capabilities and a detailed description of System Area Networks is given in section 1.1.2. Based on these features, a new approach can be made to overcome the drawbacks of existing protocol stacks. The following chapter explains which interfaces can be enhanced and how existing protocols are replaced, therefore gaining an improvement in the order of a magnitude with this new technology.

Especially its compatibility at binary level is one of its key features and will be explained in the paragraph on transparency in section 6.1.2. Hence, existing distributed applications do not need to be modified. A new compilation of the source code, or relinking of objects is not required. As a

consequence, legacy applications which could exploit raw performance of a high speed network, but are currently hampered by overburdened protocols will benefit from higher throughput.

Moreover, this middleware layer will broaden the areas of use for SAN's, which are currently limited to cluster systems where applications are parallelized by standard message passing environments such as PVM or MPI. Now, a new set of applications like transaction servers or databases will be enhanced with the potential of higher numbers of responded requests. But also applications build with other middleware layers such as the component object model (COM), the distributed component object model (DCOM) or the common object request broker architecture (CORBA) will automatically experience a performance boost as well, since at the lowest layer, they are also based on sockets.

6.1 Overview of Sockets Direct for SAN's

The following figure depicts an architectural overview on how Sockets-Direct will replace existing protocol stacks. In the traditional model a distributed

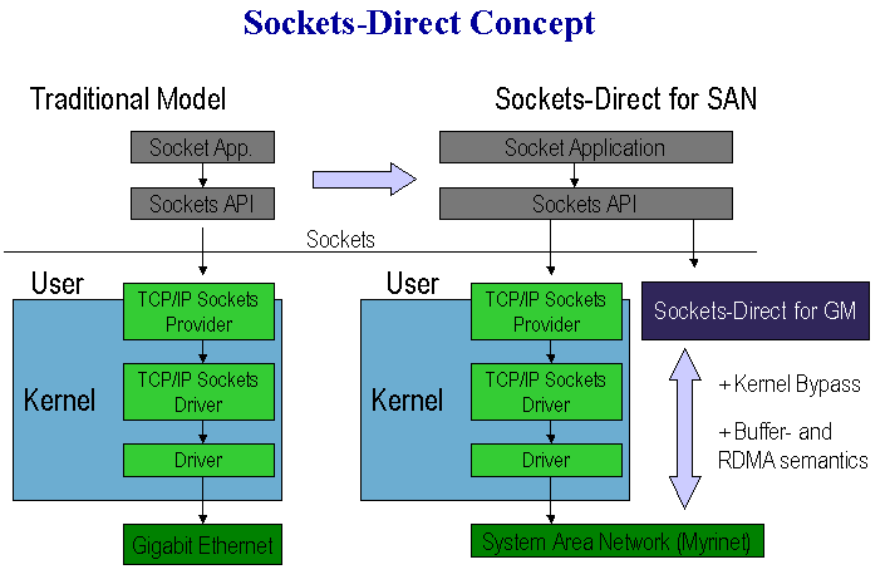

Figure 6.1: Concept of Sockets Direct

application which is using the socket interface invokes traps into the system

when calling socket *send* or *recv* functions. These traps into the operating system will *block* until the operating system has exchanged the data from user level memory to kernel memory or vice versa. Thus, when sending data the operating system has made a copy of the data, but it is not yet being delivered. When handing data to the operating system, the corresponding socket descriptor queues the data to be delivered later to the network. In order to provide message and control flow, the actual data is packaged in TCP and additional IP headers. In addition, several checksums are added and larger messages are fragmented according to the Maximum Transfer Unit (MTU) of the network. On the receiving host, the operating system has fetched and stored the data and will fill the application buffers when a receive operation becomes active. As a further consequence additional context switches occur and the performance behavior becomes worse on systems with higher work load.

When analyzing this current behavior which is the standard for any modern operating system, it is evident that user level communication using a reliable network will dramatically decrease the overhead at different levels.

When eliminating the TCP/IP protocol using a direct communication mechanism to the System Area Network, the Sockets-Direct approach as depicted in Figure 6.1 is achieved. The main advantage is kernel bypass and direct communication without additional protocol overhead using buffered copies or RDMA methods. The anticipated effects of Sockets-Direct will not only be a much higher performance, but also less CPU load for message transfers since direct control on messages will reduce the number of copies when sending and receiving data. Small packets which pass along tokens for minimal message flow will complete this setup.

6.1.1 Existing Approaches to Enhance the Performance of Distributed Applications

Current on going research tries to *optimize* the overhead of protocol stacks. One method is known as zero copy TCP/IP in which the operating system avoids the copying of data between different layers in the OSI model. In this research [63], a new feature for Solaris is developed that uses virtual memory remapping combined with checksumming support from the networking hardware, to eliminate data-touching overhead from the TCP/IP protocol stack. By implementing page remapping operations at the right level of the operating system, and caching MMU mappings to take advantage of locality of reference, significant performance gain is attained on certain hardware

platforms. This approach however requires operating system action to do the required steps. When using user level communication the operating system is no longer involved after initial setups. Similar is the work on Fbufs [22] where a a new operating system facility for I/O buffer management was implemented. This facility, called fast buffers [22], combines virtual page remapping with shared virtual memory, and exploits locality in I/O traffic to achieve high throughput without compromising protection, security, or modularity. One major drawback however is the concept itself. If pages are reused by the application again before the data has been transmitted, a costly copy on write trap will occur and the OS must prevent unsent data to be over written. This is the case for many application and no interface between kernel and application is given to test if or when the data has been sent. Moreover, this approach is also very operating system centric. It's portability on a variety of system is a time consuming effort.

6.1.2 Transparency

The replacement of protocol stacks instead of their optimization is one goal of *Sockets Direct*. However, more important are the implications for existing applications. Will it be required to modify binaries or dynamic link libraries (DLL's), or is a new compilation or additional linking required to hook the bypassed protocol into the appropriate interfaces. The available implementations of Sockets-Direct offer both features. On the one hand it is possible to link object files with a library and redefined symbols will automatically map existing socket functions to new functions with the same input/output behavior. On the other hand the concept of pre-loading dynamic libraries offers the possibility of making new functions visible before functions stored in standard libraries.

Figure 6.2 gives a detailed overview on how the interception interface looks like for all major operating systems.

Basically, the Sockets-Direct software consist of an entry point witch replaces the current available interfaces. When overwriting these entry points, the software avoids system traps but remains full control for the remainder of the code segments. As depicted, three methods can be considered as entry points for the Windows operating system. While Winsock Direct (or *Sockets Direct* Protocol currently specified by the Infiniband TA) is only available for server variants of Windows, Windows Advanced Server and Datacenter server, respectively, the concept of a layered service provider (LSP) or the binary modification using a research package name Detours [65] are available for all Windows Platforms. The implementation using an LSP is new and

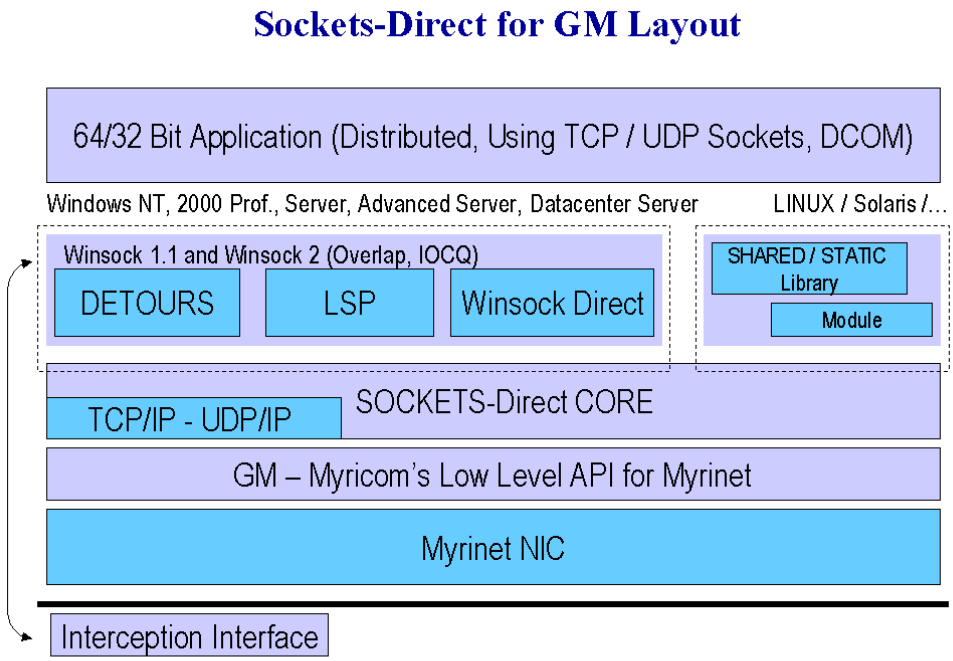

Figure 6.2: Overview on Interception Techniques

has the potential to outperform the Winsock Direct approach because the latter is abstracting from individual network interfaces. Under Unices, the concept of dynamically loading shared libraries offers binary compatibility as well. Another method which has not been implemented in detail but has been tested as a very basic prototype is possible by providing a module. The latter method has the advantage of mapping distributed applications to a high speed network by simply loading a module. In this case however, a trap into the operating system is performed. These traps can be handled very efficiently in modern operating systems and the overhead compared with calling a conventional function is only a few per cent higher [64]. For such a module, the system functions are re-mapped and the module provides a new function replacing the old function. The code segment provided in the appendix as 6.1.2 shows how this functionality can be achieved.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
/* this one contains the system call numbers __NR_... */
#include <linux/unistd.h>
/* for struct time* */
#include <linux/time.h>
/* for current macro */
#include <linux/sched.h>

/* these are macros helpfully provided for us. The utility modinfo
 * can be used to examine this stuff.
 * There are also MODULE_PARM() and MODULE_PARM_DESC macros for
 * defining and describing module parameters (there are plenty of
 * simple examples in the kernel source)
 */
MODULE_DESCRIPTION("Intercept SYS_socketcall()");
MODULE_AUTHOR("Markus Fischer, (C) 2002 GPLv2 or later");

/* we hold the old routine address in this function pointer */

static long (*sys_socketcall_saved)(int call, unsigned long *args);

static long my_sys_socketcall(int call, unsigned long *args)
{
     long ret;

     MOD_INC_USE_COUNT;

     ret = sys_socketcall_saved (call, args);

     printk("pid %ld called sys_socketcall() with call %d.\n",(long)current->pid, call);

     MOD_DEC_USE_COUNT;

     return ret;
}

int __init init_intercept(void)
{
     extern long sys_call_table[];

     /* save the old routine address, indexing into the syscall table
      */

      sys_socketcall_saved = (long (*)(int, unsigned long *))(sys_call_table[__N
R_socketcall]);

      sys_call_table[__NR_socketcall] = (long)my_sys_socketcall;

      return 0;
}

void __exit exit_intercept(void)
{
        extern long sys_call_table[];
        sys_call_table[__NR_socketcall] = (unsigned long)sys_socketcall_saved;
}

/* macros to tell module loader our init and exit routines */
module_init(init_intercept);
module_exit(exit_intercept);
```

Figure 6.3: Interception for the Linux OS.

When looking at overwriting functions the question on backward compatibility should be raised. How operating system specific tasks can be implemented and how information is gathered, which involves the operating system is another point of concern. As an example one might consider the returning value when calling the `accept()` function for establishing point to point connections between two processes. Here, the argument `addr` is a pointer to a `sockaddr` structure. This structure is filled with the address of the connecting entity, as known to the communications layer.

In order to achieve this functionality which is immediately related with the compatibility at binary level, the original functions under *Sockets Direct* are still available. Thus, for *Sockets Direct* a so called *companion socket* is created at system area network level. Preserving original functions can be achieved at different levels. When looking a shared libraries for example, the dynamic linker keeps a queue of functions with the same interface. This priority queue is build when loading shared libraries. When providing a new function which serves as a replacement for existing functions, then the `LD_PRELOAD` variable can be pointed to this new shared library and is then loaded first. Thus, the conventional function can be called by pointing to a next function with a similar interface using the `dlsym()` call. As a consequence, it is possible to provide new functions like `accept()` and `connect()` which on the one side rely on conventional functions, but which also use standard `send()` and `recv()` functions to exchange necessary information on the high speed network to allow further communication to use the low level API of the SAN directly.

6.2 Overview and Analysis of TCP/IP functionality

TCP/IP was first developed in the mid 70's by the Defense Advanced Research Agency (DARPA) to allow the communication of independent computer systems. TCP which stands for Transmission Control Protocol is a required TCP/IP standard defined in RFC 793, "Transmission Control Protocol (TCP)," that provides a reliable, connection-oriented packet delivery service. From [48] the following features on the Transmission Control Protocol can be distinguished: TCP

- Guarantees delivery of IP datagrams.
- Performs segmentation and reassembly of large blocks of data sent by programs.

- Ensures proper sequencing and ordered delivery of segmented data.
- Performs checks on the integrity of transmitted data by using checksum calculations.
- Sends positive messages depending on whether data was received successfully. By using selective acknowledgments, negative acknowledgments for data not received are also sent.
- Offers a preferred method of transport for programs that must use reliable session-based data transmission, such as client/server database and e-mail programs.

The further analysis of TCP can be described as follows: TCP is based on point-to-point communication between two network hosts. TCP receives data from programs as a stream of bytes which are grouped into segments. TCP then numbers and sequences these segments for delivery.

Before two TCP hosts can exchange data, they must first establish a session with each other. A TCP session is initialized through a process known as a three-way handshake. This process synchronizes sequence numbers and provides control information that is needed to establish a virtual connection between both hosts.

Once the initial three-way handshake completes, segments are sent and acknowledged in a sequential manner between both the sending and receiving hosts. A similar handshake process is used by TCP before closing a connection to verify that both hosts are finished sending and receiving all data.

When designing TCP, the target was wide area networks. In these environments, where packages can get lost during transmission and error rates are high, it is essential to provide mechanisms which guarantee the correct delivery of a message.

In order to run on an unreliable network with varying latencies, TCP provides flow control using the concept of sliding windows. Additionally, TCP uses a congestion avoidance technique known as *slow-start*. Upon the start of a session, a small window worth of data is sent. The sender then waits for a positive acknowledgment from the receiver and sends twice the window data, then four times, etc. until the network is saturated. This process to get to a steady state with an optimal window size is complex and time consuming.

Additional checksums to verify message integrity are added to the different layers coming with TCP/IP, also known as the stack.

Thus, for a System Area Network, the TCP/IP protocol involves a lot of overhead which reduces the maximal performance. With higher speeds in the near future, this overhead will become an even more important factor in terms of performance. The following figure depicts the overhead breakdown which has been analyzed in detail in [72], section 5.5.2.

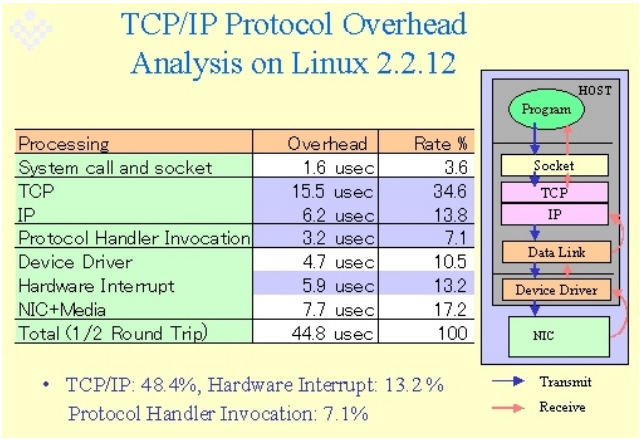

Figure 6.4: TCP/IP Overhead Breakdown [72]. About 48.4% overhead is introduced with the TCP/IP protocol. In addition 7.1% of the total time is spent in the protocol handler invocation.

Examining the overhead from several layers above the network device, it is quite evident that the stack is not optimized for a reliable network. For example, TCP uses an additional checksum for data integrity. This value has to be computed by the host CPU for every frame and is part of the overhead. Modern network hardware already have CRC's in hardware. The in-packet checksum adds additional load to the CPU because this computation is expensive. As a consequence, some Gigabit Ethernet cards have moved this computation back into the NIC.

Additional overhead comes from IP. IP splits up original TCP packages and assembles additional headers. Figures 6.5 and 6.6 give an overview of the additional protocol overhead involved with TCP over IP.

When the TCP layer passes a packet to the IP layer, it fragments the original messages using the size of the MTU. IP additionally includes fields such as 'Time to Live' to avoid data to be passed along gateways and routers continuously. These parameters are only relevant in a local or wide area network. IP also supports internetwork routing and in-flight packet frag-

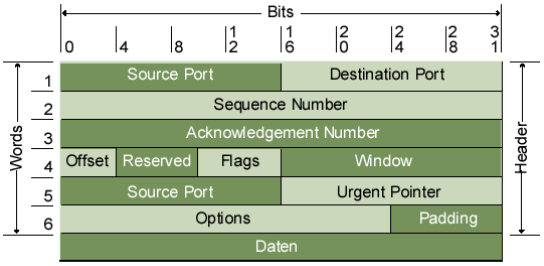

Figure 6.5: TCP Header Layout [48]

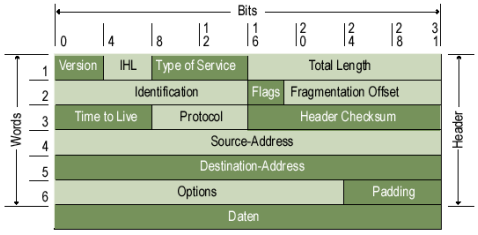

Figure 6.6: IP Header Layout [48]

mentation and reassembly, features which are not useful in a system area environment.

6.2.1 Socket Interface, Protocol and Interface Stack

Standard implementations of the Sockets interface and the TCP/IP protocol suite separate the protocol and interface stack into multiple layers. The Sockets interface is usually the topmost layer, sitting above the protocol. The protocol layer as shown in figure 6.7 may contain sub-layers: for example, the TCP protocol code sits above the IP protocol code.

Below the protocol layer is the interface layer, which communicates with the network hardware. The interface layer usually has two portions, the network programming interface, which prepares outgoing data packets, and the network device driver, which transfers data to and from the network interface card (NIC).

This multi-layer organization enables protocol stacks to be built from many combinations of protocols, programming interfaces, and network devices, but this flexibility comes at the price of performance. Layer transitions can be costly in time and programming effort. Each layer may use a different

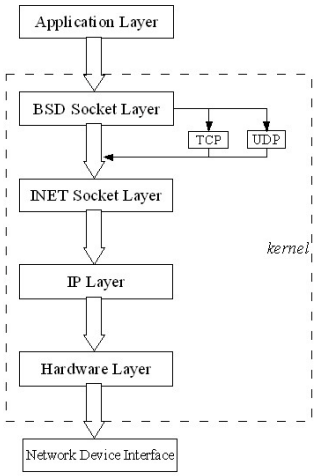

Figure 6.7: Protocol Layers extracted from the Linux Source Code.

abstraction for data storage and transfer, requiring data transformation at every layer boundary. Layering also restricts information transfer. Hidden implementation details of each layer can cause large, unforeseen impacts on performance [66]. Moreover, the raw data or actual payload is increased in its size of data actually being delivered. Adding control and header information adds several levels of encapsulation which has to be disassembled at the receiver side. This is depicted in the following figure.

Mechanisms have been proposed to overcome these difficulties when considering new generation of protocols, but existing work has focused on message throughput, rather than protocol latency [8]. With Berkeley Sockets (BSD) and the System V Transport Layer Interface, two similar programming interfaces are available. Because of their similarity the following refers to the BSD implementation only. Typically TCP/IP and UDP/IP are in use as socket protocols. If the number of layers in the communication stack could be reduced data transfer could be speeded up. This has been proposed through fbufs [22], a mechanism of avoiding data copies and switching input and output buffers in different layers. As a consequence only data pointers are passed.

In this context, figure 6.9 depicts the path through several layers from the application to the link, passing the transport and Internet layer. It also shows where data is handed to the operating system and how packets are defragmented into smaller sizes according to the MTU. Moreover, it becomes

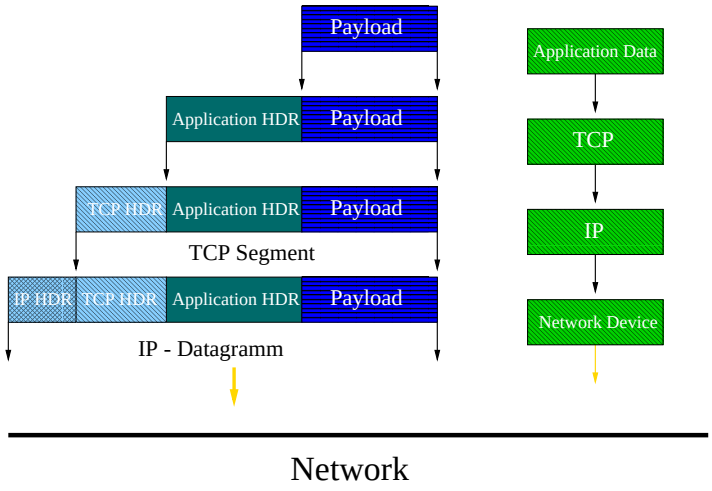

Figure 6.8: TCP/IP Protocol Encapsulation

clear, that a message does not immediately get written by the device driver but is scheduled for sending. Finally, the application which has been blocked so far returns out of the system call to continue with its program thread since at this point, the buffers can be safely reused.

6.2.2 BSD Sockets

In this paragraph an overview of BSD Sockets is given. Major points of interest are the socket interface itself and how it is suited to allow for achieving high performance in communication but also the interface level, that is, where data moves between different layers.

Depicted is the breakdown for the Linux operating system (figure 6.10). It was taken from the Linux kernel sources which are typically stored under `/usr/src/linux/`. For sending a message, BSD sockets offer three routines to operate on a socket descriptor, `write`, `send` and `sendto`, respectively. The latter one is used for communication via UDP, the others are for TCP connections established with `accept` and `connect` functions on server and client side. When calling these functions a trap into the system is made visible through `sys_*()` functions in the kernel. These calls end in the `sock_sendmsg()` function still in the BSD Socket layer before being passed to the INET socket layer. When analyzing a data exchange between two endpoints, an observation is that The standard socket model is not suited

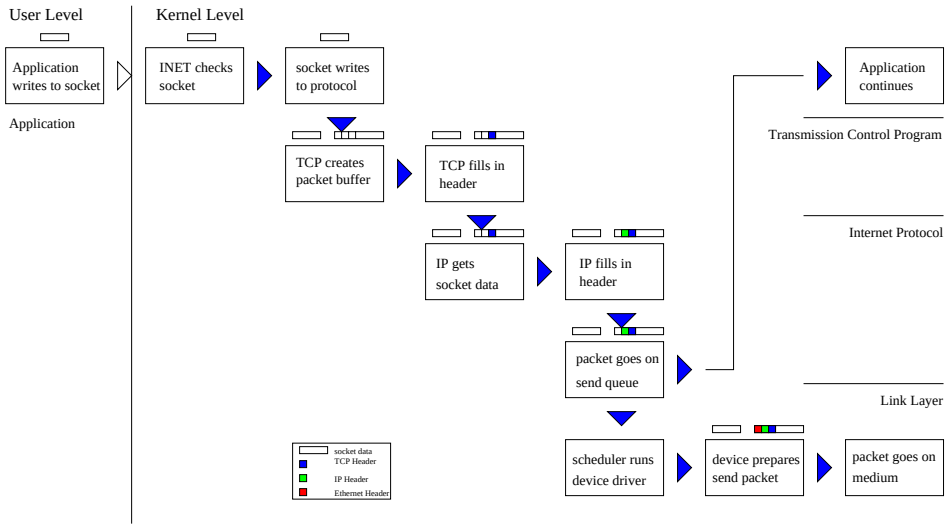

Figure 6.9: Overview TCP/IP Socket Send

for efficient implementations by design. This can be described when breaking down the different layers traversed when sending data. The process of handing data to the OS and its fragmentation into IP frames is depicted in figure 6.9. Before an application returns from a *send* operation, it spends a significant amount of time in a system call before it can continue. This time increases linearly to the message size. The reason for this is that the standard socket specification allows a message buffer to be re-used when returning from the send operation. This is very inconvenient when trying to overlap communication with computation. Since there is no synchronization between OS and application, the OS has to preserve the message data. When returning from the system call, the application can change its message buffer immediately. As a consequence, it is not easily possible to tune the BSD Socket layer for low latency. A requirement would be to make a copy of the payload immediately into an intermediate buffer before returning. Data copies however are bad when trying to offload the CPU. When describing *Sockets Direct* in full detail, the concept of buffered semantics as well as RDMA semantics using different protocols will be explained. As a requirement for an efficient streamlined implementation, the socket interface would require an asynchronous function call in which a handle is returned which can be used by the application to query the successful delivery of the data.

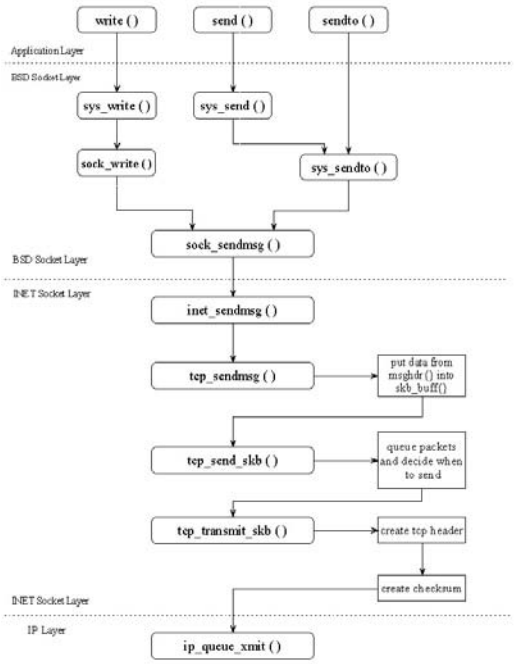

Figure 6.10: Overview BSD Send Layer

6.2.3 Winsock 1.1

The Winsock 1.1 protocol for Windows systems is very similar to the BSD Socket paradigm. It offers the same socket interface calls. One additional requirement is to initiate the usage of sockets through the `WSAStartup()` function. Socket resources are cleaned up by `WSACleanup()`. Offering the same functionality, it also inherits the same limitations. An overview for the Winsock 1.1 interface is given together with a protocol breakdown of the advanced Winsock 2 architecture.

6.2.4 Advanced Mechanisms in Winsock 2

The Winsock 2 architecture is a completely new socket interface and overcomes the major drawbacks of Winsock 1.1. It is also backwards compatible and an application can choose the protocol by specifying the `WSADATA` value to the `WSAStartup()` function. Depending on the underlying service provider, this call may be limited to Winsock 1.1 calls if a full Winsock 2 implemen-

tation is not available. The following subsections summarize the Winsock 2 features as described in [128].

6.2.4.1 Layered Protocols and Protocol Chains

Winsock 2 accommodates the notion of a layered protocol. A layered protocol is one that implements only higher level communications functions, while relying on an underlying transport stack for the actual exchange of data with a remote endpoint. An example of such a layered protocol would be a security layer that adds protocol to the connection establishment process in order to perform authentication and to establish a mutually agreed upon encryption scheme. Such a security protocol would generally require the services of an underlying reliable transport protocol such as TCP or the sequenced packet exchange (SPX). The term base protocol refers to a protocol such as TCP or SPX which is fully capable of performing data communications with a remote endpoint, and the term layered protocol is used to describe a protocol that cannot stand alone. A protocol chain is defined as one or more layered protocols strung together and anchored by a base protocol.

This stringing together of one or more layered protocols and a base protocol into chains can be accomplished by arranging for the layered protocols to support the Winsock 2 SPI at both their upper and lower edges. A special `WSAPROTOCOL_INFO` struct is created which refers to the protocol chain as a whole, and which describes the explicit order in which the layered protocols are joined. This is illustrated in Figure 6.12. Note that since only base protocols and protocol chains are directly usable by applications, only these protocols are listed when the installed protocols are enumerated with WSAEnumProtocols

6.2.4.2 Overlapping

Winsock 2 introduces overlapped I/O and requires that all transport providers support this capability. Overlapped I/O can be performed only on sockets that were created via the `WSASocket()` function with the `WSA_FLAG_OVERLAPPED` flag set, and follows the model established in Win32. Note that creating a socket with the overlapped attribute has no impact on whether a socket is currently in the blocking or non-blocking mode. Sockets created with the overlapped attribute may be used to perform overlapped I/O, and doing so does not change the blocking mode of a socket. Since overlapped I/O operations do not block, the blocking mode of a socket is irrelevant for these operations.

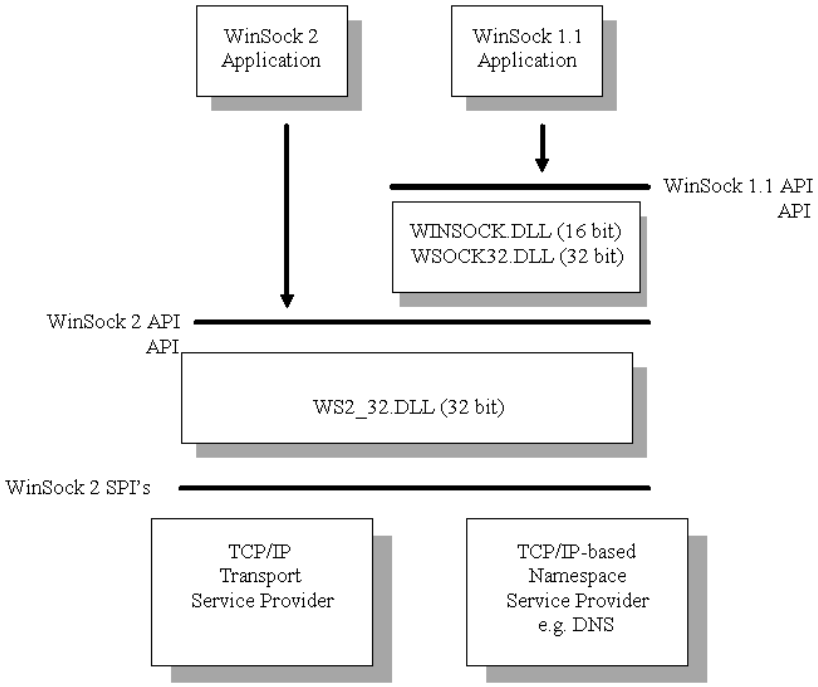

Figure 6.11: Winsock1.1 and Winsock 2 Overview

For receiving, applications use `WSARecv()` or `WSARecvFrom()` to supply buffers into which data is to be received. Winsock 2 also offers performance optimizations. If one or more buffers are posted prior to the time when data has been received by the network, it is possible that data will be placed into the user's buffers immediately as it arrives and thereby avoid the copy operation that would otherwise occur at the time the receive function is invoked. If data is already present when receive buffers are posted, it is copied immediately into the user's buffers. If data arrives when no receive buffers have been posted by the application, the network resorts to the familiar synchronous style of operation where the incoming data is buffered internally until such time as the application issues a receive call and thereby supplies a buffer into which the data may be copied. An exception to this would be if the application used `setsockopt()` to set the size of the receive buffer to zero. In this instance, reliable protocols would only allow data to be received when application buffers had been posted, and data on unreliable protocols would be lost. On the sending side, applications use `WSASend()` or `WSASendTo()` to supply pointers to filled buffers and then agree to not touch the buffers in any way until the network has consumed the buffer's contents. Overlapped

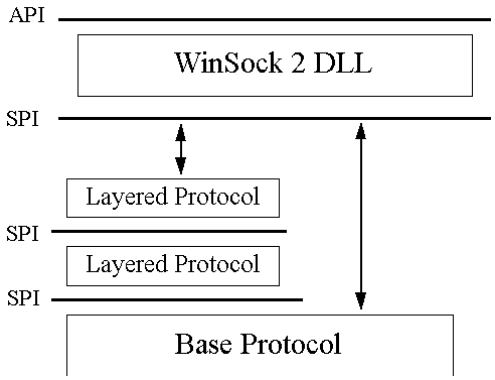

Figure 6.12: Winsock Layered Service Provider Overview

send and receive calls return immediately. A return value of zero indicates that the I/O operation completed immediately and that the corresponding completion indication has already occurred. That is, the associated event object has been signaled, or a completion routine has been queued and will be executed when the calling thread gets into the alterable wait state. A return value of `SOCKET_ERROR` coupled with an error code of `WSA_IO_PENDING` is not a socket error, but indicates that the overlapped operation has been successfully initiated and that a subsequent indication will be provided when send buffers have been consumed or when a receive operation has been completed. However, for byte stream style sockets, the completion indication occurs whenever the incoming data is exhausted, regardless of whether the buffers are fully filled. Any other error code indicates that the overlapped operation was not successfully initiated and that no completion indication will be forthcoming. Both send and receive operations can be overlapped. The receive functions may be invoked multiple times to post receive buffers in preparation for incoming data, and the send functions may be invoked multiple times to queue up multiple buffers to be sent. Note that while the application can rely upon a series of overlapped send buffers being sent in the order supplied, the corresponding completion indications may occur in a different order. Likewise, on the receiving side, buffers will be filled in the order they are supplied but the completion indications may occur in a different order.

6.2.4.3 I/O Completion Ports

Writing a high-performance server application requires implementing an efficient threading model. Having either too few or too many server threads to process client requests can lead to performance problems. For example, if a server creates a single thread to handle all requests clients can become starved since the server will be tied up processing one request at a time. Of course, a single thread could simultaneously process multiple requests, switching from one to another as I/O operations are started, but this architecture introduces significant complexity and cannot take advantage of multiprocessor systems. At the other extreme a server could create a big pool of threads so that virtually every client request is processed by a dedicated thread. This scenario usually leads to thread-thrashing, where lots of threads wake-up, perform some CPU processing, block waiting for I/O and then after request processing is completed block again waiting for a new request. If nothing else, context-switches are caused by the scheduler having to divide processor time among multiple active threads. The goal of a server is to incur as few context switches as possible by having its threads avoid unnecessary blocking, while at the same time maximizing parallelism by using multiple threads. The ideal implementation is therefore to have a thread actively servicing a client request on every processor and for those threads not to block if there are additional requests waiting when they complete a request. For this to work correctly however, there must be a way for the application to activate another thread when one processing a client request blocks on I/O (like when it reads from a file as part of the processing).

Windows NT 3.5 introduced a set of APIs that make this goal relatively easy to achieve. The APIs are centered on an object called a completion port. Applications use completion ports as the focal point for the completion of I/O associated with multiple file handles. Once a file is associated with a completion port any asynchronous I/O operations that complete on the file result in a completion packet being queued to the port. A thread can wait for any outstanding I/Os to complete on multiple files simply by waiting for a completion packet to be queued on the completion port. The Win32 API provides similar functionality with the WaitForMultipleObjects API, but the advantage that completion ports have is that concurrency, or the number of threads that an application has actively servicing client requests, is controlled with the aid of the system. When an application creates a completion port it specifies a concurrency value. This value indicates the maximum number of threads associated with the port that should be running at any given point in time. As I stated earlier, the ideal is to have one thread active at any given point in time for every processor in the system. The concurrency

value associated with a port is used by NT to control how many threads an application has active - if the number of active threads associated with a port equals the concurrency value then a thread that is waiting on the completion port will not be allowed to run. Instead, it is expected that one of the active threads will finish processing its current request and check to see if there's another packet waiting at the port - if there is then it simply grabs it and goes off to process it. When this happens there is no context switch, and the CPUs are utilized to near their full capacity.

6.2.5 Winsock Direct

Windows Sockets Direct (WSD) allows programs written for TCP/IP to transparently realize the performance advantages of user-level networks such as VIA. Programs developed to the Winsock2 API do not have to be rewritten to take advantage of changes in underlying network architecture to a SAN, nor is recompilation of these programs necessary. This enables legacy network code to work out of the box and enjoy at least some benefit of the low message latency associated with SANs. Figure 6.13 depicts a block diagram

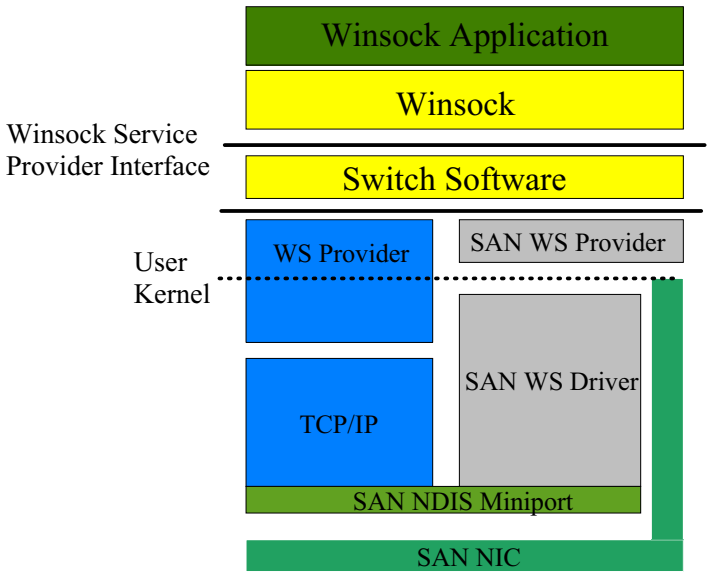

Figure 6.13: Winsock Direct Overview

of the WSD architecture. The key component of the WSD architecture is

the software switch, which is responsible for routing network operations initiated by Winsock2 API calls to either the standard TCP/IP protocol stack, or to the vendor-supplied SAN WS Provider. In addition to providing access to both of these pathways to the network on an operation-by-operation basis, the switch provides several important functions through the use of a lightweight session executed on top of the SAN provider. This session provides OOB (out of band) support, flow control, and support for the select operation. None of these mechanisms are traditionally provided by a typical SAN architecture. There are several operations that require the support of the TCP/IP protocol stack (i.e., do not use WSD), including:

- Connections to remote subnets.
- Socket creation.
- Raw sockets and UDP sockets.

Because SANs support connection-oriented reliable communication, all connectionless and uncontrolled communication must be handled by the TCP/IP protocol stack. This limits the applicability of WSD to those applications that (a) use TCP, and (b) do not make use of group communication.

6.3 Related Work

Improving communications performance has been a popular research topic for a long time. Previous work has focused on protocols, protocol and infrastructure implementations, and the underlying network device software. But also TCP/IP stack implementations have been ported to other networks. In the following an overview on related work will be presented.

6.3.1 Streamlined Socket Interfaces

Early work [17] analyzed communication overhead and argued that protocol implementations, rather than protocol designs, were to blame for poor performance. The author stated that an efficient implementation of a general-purpose protocol would allow for the same performance as a special-purpose protocol for most applications. The work presented in [67] found that memory operations and operating system overheads played a dominant role in the cost of large packets. For small packets, however, protocol costs were

significant, amounting for up to 33% of processing time for single-byte messages. The x-kernel presented in [45] was an OS targeted at high-performance communications by reducing system call costs. Later work [21] [24] focused on hardware design issues relating to network communication and the use of software techniques to exploit hardware features. Key contributions from this work were the concepts of application device channels (ADC), which provide protected user-level access to a network device, and fbufs [22], which provide a mechanism for rapid transfer of data from the network subsystem to the user application.

The development of a zero-copy TCP stack in Solaris [63] avoided memory copies when traversing the stack. By using direct memory access and operating system features such as page re-mapping, and copy-on-write pages to improve communications performance. A page marked with Copy-on-write avoided a memory copy if the application did not modify the data before the data was delivered. When doing so, an interrupt would occur and the OS would have to make a copy of the page before the application could proceed. Moreover user applications had to use page-aligned buffers and transfer sizes larger than a page to gain best efficency with the zero protocol stack. The resulting system achieved reasonable throughput for large transfers (16K bytes), however smaller packets that make up the majority of network traffic did not gain better performance.

An alternative to reducing internal operating system costs is to bypass the operating system. This results in a user-level library like *Sockets Direct*.

Earlier work [40] and [54] explored building TCP into a user-level library linked with existing applications. Both systems, however, attempted only to match in-kernel performance, rather than its improvement. Further, both systems utilized in-kernel facilities for message transmission, limiting the possible performance improvement. Similar work was carried out in [25]. It was an entirely user-level solution which replicated the organization of the kernel. The performance however was worse than the in-kernel TCP stack.

[67] showed that interfacing to the network card itself was a major cost for small packets. Recent work has focused on reducing this portion of the protocol cost, and on utilizing the message coprocessors that are appearing on high-performance network controllers such as Myrinet [51].

Other work on implementing a stream sockets layer was conducted in the SHRIMP project. SHRIMP supports communication via shared memory and the execution of handler functions on data arrival. The SHRIMP network had very low hardware latencies. It used a custom-designed network interface for its memory-mapped communication model. Using sender-based memory

management, the bandwidth of SHRIMP sockets was only about half the raw capacity of the network hardware, but it also dealt poorly with non-word-aligned data. For unmodified applications, this introduced additional programming complexity to work around.

The U-Net described in 2.1.3.3 was also supported by a TCP implementation (U-Net TCP). This protocol stack provided full functionality of the standard TCP stack. With a modification for better performance it succeeded in delivering the full bandwidth of the underlying network but still imposed more than 100 microseconds of packet processing overhead relative to the raw U-Net interface.

Fast Sockets is another implementation of the Sockets API using Active Messages as a transport mechanism that provides high-performance communication and inter-operability with existing programs. It was most recently described in [69]. Interoperability with existing programs is obtained by supporting most of the Sockets API and transparently using existing protocols for communication with other sockets programs not using Fast Sockets. In order to let an application use Fast Sockets, applications must be re-linked. Moreover, Fast Sockets cannot currently be shared between two processes for example, via a `fork()` call. An application using `exec` will lose all Fast Sockets states as well. This poses problems for traditional Internet server daemons such as inetd, which issue a call to `fork()` for each incoming request.

The Fast Socket implementation also faced problems on process termination. Gracefully shut downs on sockets could not be handled.

Just recently the efficient implementation of a socket interface has become popular again. The emerging Infiniband standard introduces a set of new protocols and a socket interface which involves only low host utilization is under specification. With Sockets over VIA (SOVIA) [70] a new socket implementation using the VIA interface was presented. This implementation which is only available for the Linux OS requires source code modification in order to let an application use the VIA interface directly. Several other limitations exist in SOVIA. For example, the handling of `fork()` is not supported. The lightweight sockets project from Hitachi [71] can also be seen as a recent investigation on streamlined interfaces. The software is suited for smaller client server examples. Full semantics are not provided.

6.3.2 Suitability of the Transmission Control Protocol for System Area Networks

When deploying TCP/IP on top of System Area Networks, the complete protocol stack is implemented. This results in better communication performance due to the higher performance of the physical layer. Its round-trip latency is still poor. However, a comparison with the performance of the low level API of the SAN shows a large performance gap. The latency of a current high speed network is less than 10 micro seconds, however the costly system trap which comes with a stack implementation of TCP/IP will involve much higher values.

Another factor is less bandwidth when applying a TCP/IP stack on top of a SAN. This is due to the TCP/IP concept in general and several aspects of TCP/IP and their impact on performance will be discussed in the following. A performance chart in Figure 6.14 will give a first look on how the TCP/IP protocol will degrade available performance.

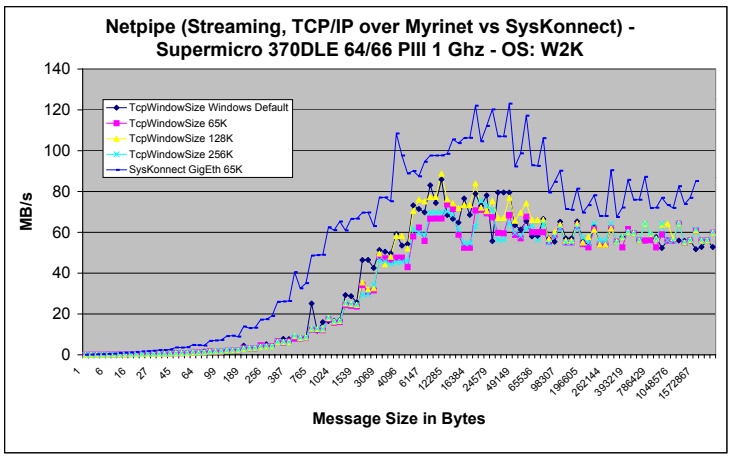

Figure 6.14: TCP/IP Performance Comparison on Myrinet and Syskonnect using Netpipe

A low level API for Myrinet will typically peak at 247MBytes/s, reaching this value for 4096 Bytes. Half of this Bandwidth is available using a message size of 1300 Bytes. The tuned TCP/IP protocol delivers a maximum of

87MBytes/s for a message size of 9000 Bytes. Half of this bandwidth is reached for messages with approximately 2000 Bytes. Another discrepancy is the latency. While GM has a one way latency of 8 micro seconds, the TCP/IP over GM implementation has a one way latency of 75 micro seconds.

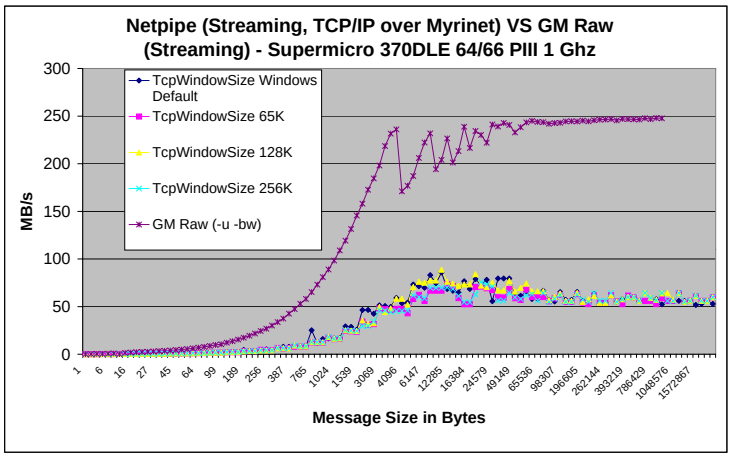

Figure 6.15: Performance Comparison of GM over Myrinet and Netpipe over TCP/IP over Myrinet

Figure 6.15 indicates that much of the available performance is lost due to the overburdened TCP/IP protocol stack. While the SysKonnect Gigabit Ethernet card achieves 90% of the raw performance, the TCP/IP stack over Myrinet only achieves 1/3 of the available performance of the low level API GM.

Since Gigabit Ethernet networks compete with System Area Networks, its performance is included as well. We have tested a high end Gigabit Ethernet network card from SysKonnect. The SysKonnect card which supports Jumbo frames of 9000 Bytes reaches almost wire speed which is 125 MBytes/s. In comparison to other Ethernet cards operating in Gigabit mode, this is an outstanding implementation. A large fraction performs rather poor [78].

These results lead to the conclusion that TCP/IP can achieve better throughput on faster networks, its round-trip latency however remains poor. Next-generation network technologies like Myrinet do not begin to approach

the raw capabilities of these networks [5]. In the following we describe a number of features and problems of available TCP implementations, and how these features affect communication performance.

6.3.2.1 Targeting Local-, Wide- and System Area Networks

Following the description on TCP in section 6.2, TCP/IP was originally designed, and is usually implemented, for wide-area networks. Since several entries which are useful in error prone environments are no longer needed, the stack is undergoing several optimization techniques. For example, TCP uses an in-packet checksum for end-to-end reliability, which can cause high CPU load and can create a bottleneck when a stream of packets needs to be handled. Modern system area networks however come with a per-packet CRC. Other parameters like the Time To Live field in the IP header are not relevant any more but increase message sizes. IP also supports internetwork routing and in-flight packet fragmentation and reassembly, features which are not useful given the new features.

6.3.2.2 Stack Implementation Details

Standard implementations of the Sockets interface and the TCP/IP protocol suite separate the protocol and interface stack into multiple layers. According to figure 6.10, the sockets interface is the first layer accessed by a socket application. The protocol stack then contains several sub-layers, which need to be traversed. Finally, the interface layer communicates with the network hardware. It is separated by the network programming interface, which prepares outgoing data packets, and the network device driver, which transfers data to and from the network interface card (NIC). This organization allows flexible combinations but restricts room on efficiency and optimization. Performance can be degraded since transitions are costly. Different layer typically do not represent data in the same format and the actual message is encapsulated or fragmented at different layers. Moreover, memory management is not visible across different layers. Hence, the data is copied from layer to layer causing very high CPU load.

Mechanisms have been proposed to overcome these difficulties [6], but existing work has focused on message throughput, rather than protocol latency [8]. As a consequence, current TCP/IP implementations required a complicated memory management mechanism. In a multi-layered protocol stack packet headers are added (or removed) as the packet moves downward (or upward) through the stack. With the lack of interoperability, this requires

additional memory copies. Buffer mechanisms inside the operating system kernel however are limited. To overcome some of the drawbacks, BSD introduced the concept of mbuf. An mbuf can directly hold a small amount of data, and mbufs can be chained to manage larger data sets. Chaining makes adding and removing packet headers easy. The mbuf abstraction is not cheap, however: 15% of the processing time for small TCP packets is consumed by mbuf management [67]. This technique however does not avoid memory copies; user data must be copied into and out of mbufs. Using mbufs, nearly one-quarter of the small-packet processing time in a TCP/IP stack is spent on memory management issues. As a consequence the overhead introduced with complicated memory management is therefore a viable target to improve communications performance. This is the subject for a direct sockets implementation which will be described in the following.

6.4 Design Space for Sockets Direct

The design space for a *Sockets Direct* implementation is dependent on the final goal the middleware layer is aiming at. The complexity varies from small server client applications each consisting of simple `send()` and `recv` functions to applications which use a set of functions like `fork()`, `dup()`, `dup2()` and `close()` on socket descriptors.

Moreover the complexity is defined by supporting legacy applications which have to work right out of the box, or if application are even allowed to be modified at source code level. The following will describe available design spaces and their trade off.

6.4.1 Sockets Direct Portability among Major Operating Systems

Given the widespread usage of TCP/IP and UDP/IP application, the portability of *Sockets Direct* is of importance. In this section platform dependent interfaces will be examined. Providing portability requires deep knowledge of operating system specific features. It also requires the understanding of how applications are build today.

6.4.1.1 Replacing standard libraries

When compiling and linking a program, it can be either statically or dynamically linked. Static linking has the advantage that the code can be easily

distributed. The program in binary format contains all necessary code segments to be started. For *Sockets Direct* this would mean that a standard library which contains the basic transport provider for the socket interface must be modified. This would however require to replace current libraries. It has been proven that any modification to a standard system will most likely avoid the acceptance of a project in general. This would be true for replacing standard libraries as well as patching an operating system in order to achieve the aimed functionality. Therefore any project must be designed not to interfere with the basic system. Thus, only an extension like a *module* or a library as an add on to existing libraries are a feasible approach. For static linking, *Sockets Direct* offers an additional library. With advanced linker functions, it has been made possible to remap functions, while preserving access to original functions as well. The following sequence shows how this can be done:

```
ar xv /usr/lib/libc.a send.o recv.o
ld -r -x -defsym SD_TRAMP_send=__send \
         -defsym SD_TRAMP_recv=__recv \
 recv.o send.o -o libsocketsdirect.o
```

In this case, the code from the original `send` and `recv` routines are preserved and accessible through calls to `SD_TRAMP_send` and `SD_TRAMP_recv`, respectively. The *Sockets Direct* middleware layer can now provide its own send and recv functions and can rely on the trampoline functions. When linking an application statically, the linker will remap functions when providing the libsocketsdirect.o library.

6.4.1.2 Providing shared libraries

Static linking however also has some disadvantages. When linking all required objects together, executables can become extremely large. Although memory has become cheap, the executable will take more memory as if a dynamic linking would be used. The latter offers the possibility to share code among processes. The text segment will be accessible (shared) for reading by multiple processes given their own data segment. This is very efficient since a large portion of code is similar since high abstraction libraries are used.

```
g++ -o sockets-direct-unix.so -fPIC -c sockets-direct-unix.cpp
g++ -shared -Wl,-soname,libsockets-direct.so.1 \
 -o libsockets-direct.so.1.0 sockets-direct.so *.so
```

When building shared libraries, then the -fPIC flag generates position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. All constant addresses are stored in a global offset table. When starting an application, the dynamic loader which is part of the operating system resolves the entries in the global offset table. It will create a queue of references to functions with the same interface. When a function 'f' is called, then its first reference will be executed. Modern operating systems like Linux or Solaris offer a pre-loading function. Typically, the `ldd` program will print the reference list for an application. When using the pre-loading feature, then the shared library which was specified will be referenced first. This concept is a viable method to implement *Sockets Direct*. Shared libraries can be loaded upon request, their pre load however can be specified permanently as well (`/etc/pre.load`). One important feature is to be able to walk through the reference list of the dynamic linker. The `dlsym` function call can provide a pointer to the next reference in the chain. It should be noted, that the feature of pre loading shared libraries guarantees binary compatibility, given that full semantics are provided by the favored *Sockets Direct* shared library. This will make it possible to map legacy applications to use *Sockets Direct* as their communication layer.

6.4.1.3 Binary Instrumentation

Another way for mapping legacy applications to a different protocol is that of binary instrumentation. Tools like Detours [65] for Windows and SLI [80] for Solaris enable interposition aside of any object middleware. In Detours, function calls are dynamically intercepted by re-writing function images in order to redirect the control flow to different locations. In contrast, SLI interposition is based on symbol preemption in the resolution mechanism. In this way, SLI can dynamically introduce profiling and tracing functionality into dynamically-linked programs without changing the program image.

6.4.1.4 Layered Service Providers

The concept of a layered service provider is a rather new mechanism and was introduced with the Windows NT 3.51 operating system in 1994. Any layering technique will require the insertion of new protocols into a chain which lists all available protocols. This way, additional services like Quality of Service (QoS) may be installed above a basic transport provider. So far this technique has not been adopted to any UNIX flavored operating system such as LINUX or Solaris.

6.4.1.5 Winsock Direct

Microsoft has provided a switch which will map socket functions to use the System Area Network when available. Winsock Direct abstracts from a SAN and offers a general interface. Each SAN vendor must provide its own interface for Winsock Direct according to the Service Provider Interface (SPI). For this, a service provider DLL must be written and registered with the operating system. In effect, this new protocol fits seamlessly underneath the Winsock API that serves to integrate server applications into SAN environments. Winsock Direct bypasses the kernel networking layers and communicates directly with the SAN hardware. Winsock Direct is available for Server Systems only. It was introduced with Microsoft Windows 2000 Datacenter Server and is now available as well in Windows 2000 Advanced Server having Service Pack 2 installed. *Sockets Direct* was not implemented under Winsock Direct, however its interface would be an alternative method for an implementation. However, the performance might be less than the available performance given from low level API's of the SAN [81]. This is most likely due to the general approach to serve any SAN.

6.5 Sockets Direct Implementation

In the following details on a *Sockets Direct* implementation will be presented. This work was conducted using the GM over Myrinet API as a reference implementation. No restrictions are given when using other system area networks. If relevant implementation details according to the GM interface were used, this will be stated explicitely.

The implementation was done based on providing a shared library in the Solaris/Linux environment which allows for binary compatibility, a static library which requires the relinking of object code. For the Windows environment, the full implementation was performed for binary interception techniques as well as a layered service provider. The latter one achieves binary compatibility. Binary interception requires the modification of the executable using a tool coming with the Detours package [65].

6.5.1 Setup and Connection Management

The sockets interface provides a variety of socket calls. However only 15 per cent of them are related to exchanging data. For the *Sockets Direct* implementation the setup functions to let further communication use the low

level API of a System Area Network are expensive. However, this setup procedure is performed only once, In those setup functions the *Sockets Direct* implementation relies on basic TCP/IP behavior. This way, a traditional connection is created between two endpoints, but also the SAN connection is set up. This mechanism also offers a flexibility which is required for different SAN types. Some of them require an explicit connection between two endpoints (like VIA or ATOLL), others are connectionless (like GM over Myrinet) for example. As stated earlier, the approach of a *Sockets Direct* implementation is available for several operating systems, including different interception levels. The following description will hold for any variant of *Sockets Direct* unless stated otherwise. The implementation is explained in more detail in the following.

6.5.1.1 Companion Socket Initialization in Sockets Direct

The number of ports which are exposed to applications is still limited, even on popular high speed networks. Therefore the implementation of *Sockets Direct* has a very resource conservative approach. It only requests an endpoint to be mapped into user address space, when the application is actually using the socket interface. Prior to any communication, two socket endpoints need to establish a point to point connection. While Windows Sockets require the `WSAStartup()` routine to be called prior to calling any other Windows Socket function, this is not the case for any other socket implementation. Moreover, a socket application indicates which Winsock version it would like to use, when calling `WSAStartup()`. For *Sockets Direct*, this routine is intercepted and buffer management and other internal structures are initialized. Prearrangements to be Winsock 1.1 or Winsock 2 compliant can be made through the argument parameter provided to the WSAStartup call. Since other socket implementations lack the support of a startup function, the `socket()` call has been interfaced. When examining the socket interface API, this is the basic function which is called to create a socket which can be later on used in send or receive functions.

6.5.1.2 Establishing Connections

The TCP protocol only supports point to point connections which are established through `accept` and `connect` calls.

When intercepting these functions, a relation between socket descriptor and corresponding information of the system area network is stored. This relation is then used when exchanging data. More precisely, when establishing

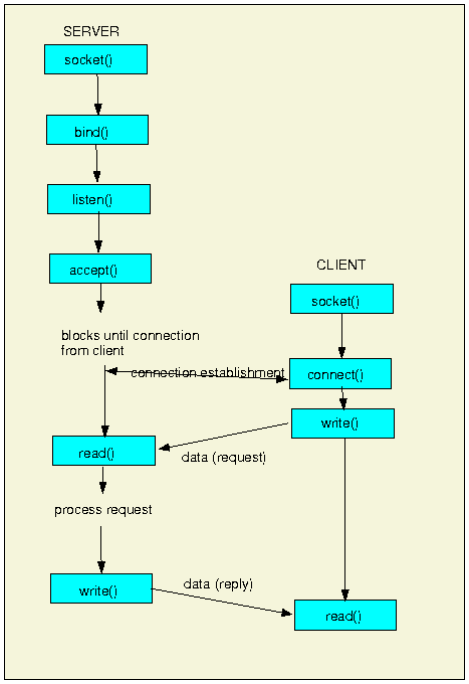

Figure 6.16: TCP Socket Connection Establishment

connections, a so called companion connection is made. For a server which calls `accept()` to serve its clients, the *Sockets Direct* implementation first lets the TCP/IP `accept` call succeed which returns evident information in the socket structures. With them it is possible to figure out which host(name) issued the connection. Figure 6.16 depicts a typical connection sequence in which a server accepts clients to allow for data exchanges based on a point to point connection.

In the reference implementation, the GM lookup function `gm_hostname_to_id()` can be used to determine whether the client is a potential candidate for a companion socket. If this is the case an additional hand shake inside of the accept and connect sequence will store a reference of the socket descriptor with a target node id, a target port id and crucial for multiple socket connections between the same nodes, a unique socket identifier. The latter one is determined to be the socket descriptor itself.

This setup mechanism holds for Winsock 1.1 and Winsock 2, but also other socket implementations. For the client, the `connect` call is even simpler

to handle. The argument in the `connect` function provides the hostname.

6.5.1.3 Sending and Receiving Messages in Sockets Direct

Two variants for an exchange model which are dependent on the message size are implemented in *Sockets Direct*. First is that of buffered copies. When sending messages, the message gets fragmented and copied into registered buffers of GM. Finally, the GM send function `gm_send_with_callback()` is invoked, which will activate the DMA engines to inject the messages into the network. With this message handler in hardware, the send functionality can be easily achieved. On the receiver side, a matching `gm_receive()` will allow data to be copied from the registered buffers into application buffers. Earlier, data has been copied into registered buffers by network-to-host DMA engines.

Using this mechanism, *Sockets Direct* is compliant with any socket interface. For the Winsock 2 architecture, a posted overlapped receive requires additional threads which handle incoming messages.

Figure 6.17 depicts an overview of available models in Sockets Direct. Basically one can distinguish between a buffered model and a zero copy protocol. While the buffered model allows for very low latency and minimizes the function call overhead, the zero copy protocol aims at offloading the host CPU. Using the buffered model, the function call overhead is reduced since a copy of the message is created. It is the responsibility of underlying SAN API to guarantee the correct delivery. Since the SAN API will be active based on intervals (the API is called, or an additional thread watches the communication ACK's), it is possible to either retransmit the message or free the buffer upon successful message delivery.

Figure 6.18 presents the buffered semantic in more detail. The buffered protocol will provide best performance for small messages, but can be used for larger messages as well. The latter are fragmented into sizes which fit the size of the private buffer pool bins. The destination will then concatenate the message again. When using the buffered model, the overhead of the calling application when sending a message will be very small. The data will be copied into the private buffer pool and the original buffer can be used again. Especially when streaming data from one process to another, this model will effectively saturate the network since this model reflects a pipelining strategy. The performance enhancements through pipelining were presented in section 3.3.5. The zero copy protocol using *writes* which aims at off-loading the CPU when communication between two processes is requested, can gain efficiency

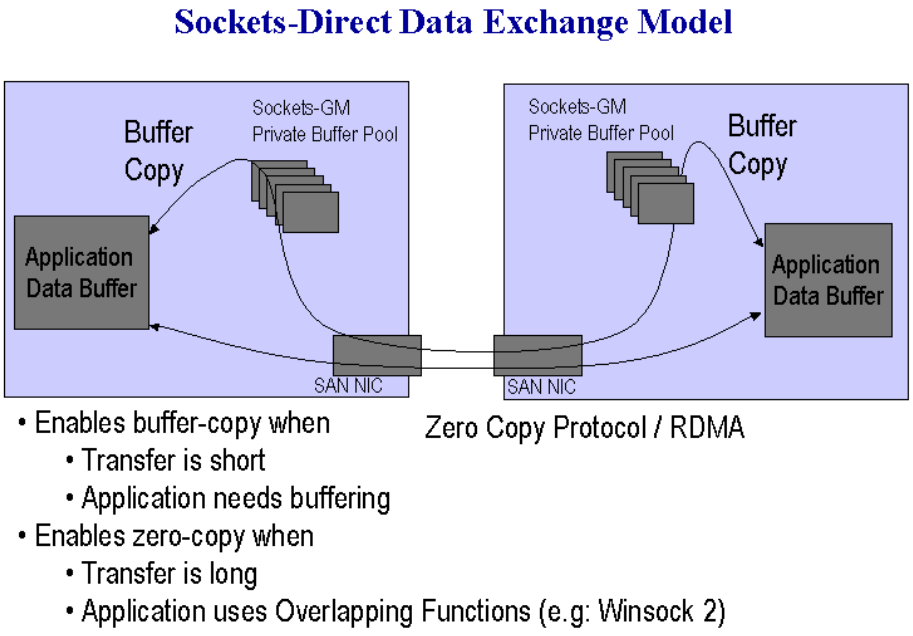

Figure 6.17: Overview Sockets Direct Transfer Exchange Model

only for larger messages. This is because the exchange of data requires a short rendezvous prior to sending the payload. The advantage is that the payload is deposited directly into the application buffer and no additional copy is required. Thus, the rendezvous provides the virtual address on the receiver side. In addition to providing the virtual address which can be used for a remote store, the receiver must achieve a virtual to physical address translation at the NIC. A more detailed description on this topic can be found in chapter 5. Another zero copy protocol can be implemented which *reads* the data from the origin. This is also called a *get* method, in which the sender only provides the location of the payload. If the receiver posts a receive and detects a message, this message will only provide a description where to find the data in the address space of the remote process. The receiver can then *get* the data into its application buffer. This transaction is then finalized by informing the sender with a short message to release the buffer. This protocol must guarantee that the sender does not modify the data. When looking at the socket interface of BSD or Winsock 1.1 as it

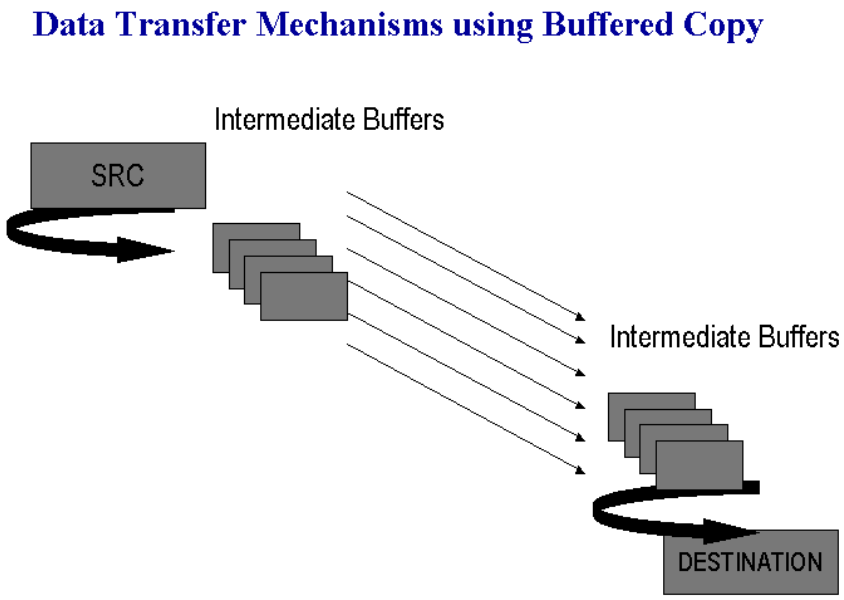

Figure 6.18: Sockets Direct Transfer using Buffering Semantics

specified today there is no solution to implement this efficiently.

6.5.1.4 De-multiplexing of Incoming Messages in Sockets Direct

The GM API over Myrinet only provides a single receive queue, which can dequeue only the head of the queue. When operating on sockets, messages can be received and detected on different file descriptors. Thus, it is not sufficient to read the head of the queue when receiving on a certain file descriptor. It could be possible that the following messages in the queue are destined to reach a different socket descriptor.

Another limitation would be in not supporting overlapped structures under Winsock2 since the GM interface is not designed to autonomously dispatch incoming messages to posted structures.

Thus, the de-multiplexing of incoming messages is handled by a Worker thread which will dispatch incoming messages and insert the messages into a receive queue for every established point to point connection.

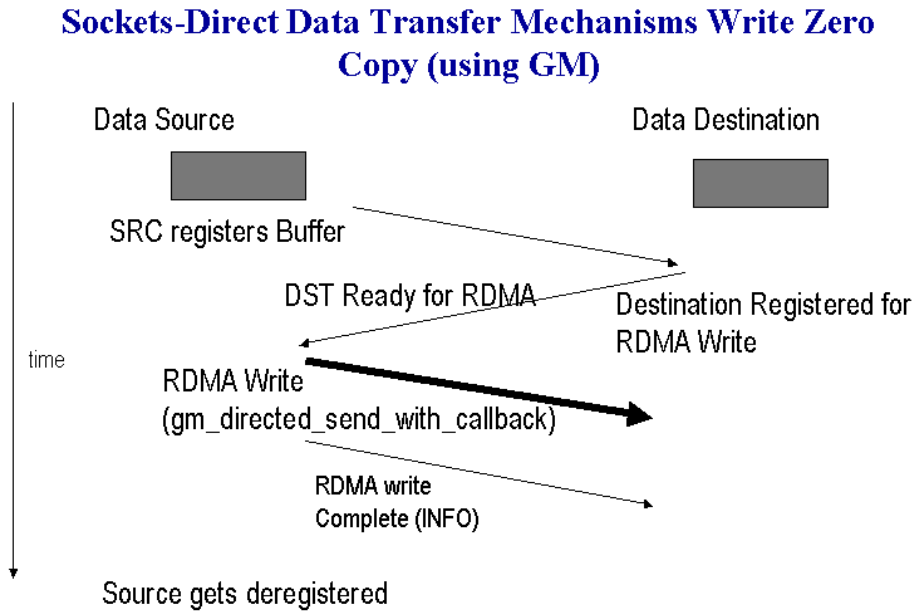

Figure 6.19: Sockets Direct Transfer using Write Zero Copy

To be Winsock2 compliant, two receive queues exist for each socket descriptor. One stores incoming messages, which have not yet received a matching receive call from the application. The other queue stores pending overlapped receive operations.

When implementing *Sockets Direct*, one multi-threaded variant for demultiplexing messages was targeted at SMP systems. In this implementation, one thread would only query internal data structures, while another worker thread would service the network and move incoming data to the corresponding queue. This however resulted in very poor performance due to the required synchronization.

In the standard TCP/IP implementation, the operating system serves as a dispatching instance and handles incoming data with a system buffer, which is limited in size. Once the buffer is full, sending further messages will block. For *Sockets Direct*, messages are taken from the network and DMA'd into provided buffers. Once these buffers have been occupied, the MCP will stall on delivering data. Thus it is required to move data, even if the destination

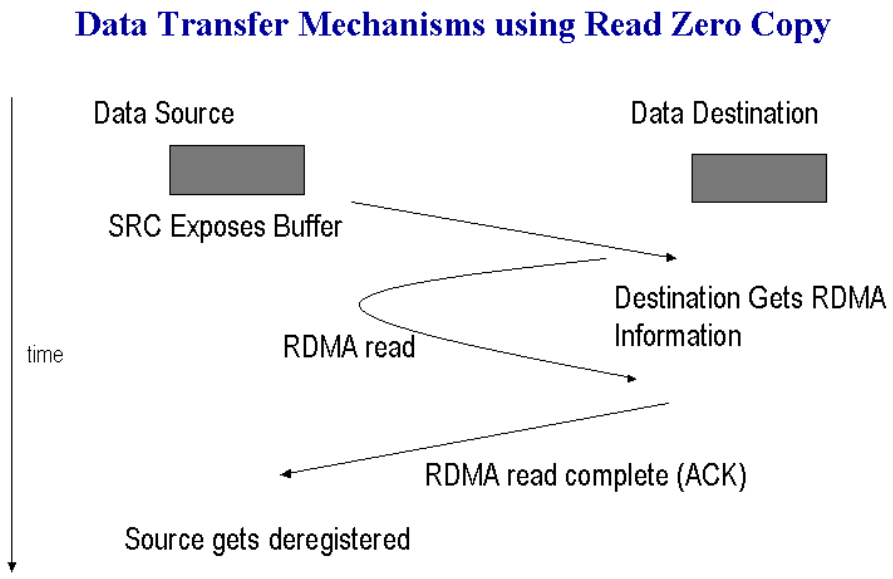

Figure 6.20: Sockets Direct Transfer using Read Zero Copy

does not post receive for a while. For this, additional `malloc()` calls are required to provide further memory space to the application. This memory is freed again when the application does a receive.

This strategy however, requires a costly notification through events when messages come in and the application already waits for data. But also, several calls to `malloc()` and `free()` slow down the performance and an additional memcpy between worker thread and the data segments of the application is required. An advanced approach is to avoid notification through events and additional memcopies.

This can be achieved when performing the receive on GM structures by the application itself. When invoking the socket `recv()` function, only one descriptor is queried. The idea is to take any message out of GM's message queue and return when an appropriate message has been found. The reason for this strategy is that in most applications there are service connections and connections which are used heavily for data exchanges. Under these circumstances, the possibility of immediately finding the message waiting for

is high and other incoming messages are enqueued for later usage. To be compliant to the socket interface in general, one additional thread however will issue receives itself when the main application did not perform receives for a while. This guarantees, that messages are taken from the network. It also does not affect performance, since the application did not request the data for a while. The sender however can continue its cycle. As a result, this mechanism also works for overlapping concepts in Winsock 2. In benchmarks, the second strategy achieved 30 per cent improvement over the worker thread model.

6.5.1.5 Establishing Full Semantics, Sockets Direct Limitations

One of the challenges in providing a user level socket implementation is that of being fully functional against the TCP/IP implementation. This also requires the handling of abnormal termination. For example, when a control-C or SEGV occurs, the communication partner has to be notified. In the standard TCP/IP implementation the operating system serves as a central instance of control. Using a user level communication principle, additional functions have to be implemented in order to achieve the same behavior. The socket directs implementation achieves full semantics, even for segmentation faults, as well as immediate kills while keeping high performance for standard situations. Standard situations are given when a single point to point has been established. When using function calls like `fork()`, all sockets of a process become shared. This introduces additional control tokens for handling this situation because current low level APIs of user level networks do not support shared accesses to a single port. In this situation, the traditional socket protocol is incompatible with point to point network protocols and extra software is required to overcome this situation. Typically one process (the parent) closes sockets which are no longer needed. This way, a direct point to point connection can be established again and a match for socket descriptor and recipient is bijective again. Thus, there are no limitations in *Sockets Direct*.

The internal data structures can be better understood by providing the

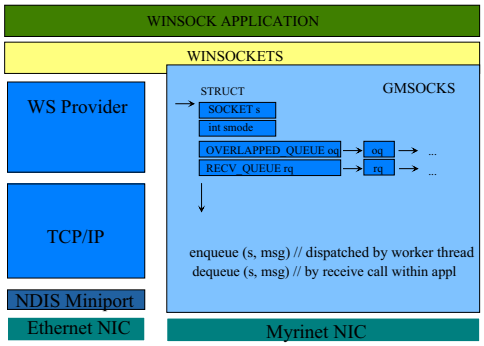

Figure 6.21: Sockets Directs Internal Data Structures

```
typedef struct gmsock_con_t_ {
   SOCKET s;

   int s_close_init;
   int msg_id;

   // list of descriptors which represent the same socket.
   // see dup()
   list <int> dupped_list;
   queue <queued_message_for_shared_socket_t> qm;
   queue <requested_message_for_shared_socket_t> rm;
   int remote_s;           // partner's socket descriptor

   // variables which will help handling shared socket stati
   int sock_blocked;    // we can not send data on SAN
                        // since remote process calls fork
   int sock_shared;     // when the socked is shared - value > 0,
                        // do not use SAN as transport
   int sock_wait_for_ack;
   int connect_req_status;

   // socket states
   int allow_send;   // support for shutdown
   int allow_recv;   // support for shutdown
   int sock_closed;  // EOF received
   int sock_closed_ack;  // waiting for ACK after sending EOF
   int sock_failed;
```

```
   long t_keep_alive_send;  // the time we send an alive for/on
                            // socket s
   long t_keep_alive_recv;  // the time we received an alive
                            // for/on socket s

   // SAN specific information to determine communication partners
   int gmtid;
   int gmpid[GMSOCK_MAX_GM_PORT];  // [i] = 1: port i is
                                   // communciation partner
   socket_recv_queue grecv_q;      // not yet matched messages to
                                   // be received
} gmsock_con_t;
```

For overlapping operations under Winsock 2, a receive will dequeue pending messages out of the linked receive queue. Otherwise it will enqueue the overlapped request and the worker thread will dispatch an incoming message into the provided overlapped struct and signal its completion.

6.5.2 Optimizations through Winsock2 Overlapping Mechanisms

The Winsock 2 architecture is the most advanced socket interface implementation as of today. It is suited very well to write scalable applications, but leaves many possibilities to be implemented very efficiently. This is also true for the *Sockets Direct* implementation. Using overlapping operations for example, an application can post send or receive operations and query their status later. In the meantime, an application can perform time consuming task but let the data exchanges be handled in the back. Typically the operating system will handle these requests, but user level communication must take care of these operation itself. It is therefore of importance to have active worker threads which service the network, but also mimics operating system features. This is when the advanced features of the Winsock 2 architecture also challenge a *Sockets Direct* implementation. When handling overlapped operations a worker thread must either progress on injecting data into or retrieving data from the network. This must be handled autonomously and efficiently. Moreover the most efficient strategy which serves the network itself, but relies on a worker thread for compliance must synchronize multiple threads safely. When posting a overlapping receive, the worker thread must fill posted buffers with incoming data

and signal internal events which are queried from time to time by the application itself using `WSAGetOverlappedResult()`. The problem is worse than for example in implementing this behavior in MPI, since the call to `WSAGetOverlappedResult()` is not accessible by a *Sockets Direct* implementation. The same applies when a scalable application uses a Windows centric concept of I/O completion queues. These queues signal the application if a operation on a socket is available. This can be for example a pending send or recv. To allow for very efficient process handling, this is an important feature. The *Sockets Direct* implementation is capable of fulfilling this request by calling WPUCompleteOverlappedResult on assorted queues.

6.5.3 Impact of User Level Mode vs Kernel Handling

The traditional TCP/IP protocol will use system calls to exchange messages. The operating system kernel will then be responsible for data transmission. It will also detect operations on a socket which are not data transfers. A socket therefore describes an interface with the OS. Starting from its creation through the call to `socket()`, a socket descriptor has different states. Figure 6.22 depicts these states as defined in [48].

The socket direct implementation has to keep these states in mind. As a user level implementation, the *Sockets Direct* implementation has no direct access to its states, thus is required to keep a copy itself. Some operations on sockets also do not have an influence on the *Sockets Direct* behavior. This would for example be a setting to socket modes or the socket buffers.

6.5.4 Process Management, Shared Sockets and Exception Handling

The chosen approach in *Sockets Direct* which uses a companion socket is ideal for achieving binary compatibility. Companion socket in this context means the creation of a traditional socket and a connection at SAN level. First, the traditional establishment using a pair of accept and connects will allow a using the traditional protocol to be used for exchanging information to setup a connection. This connection remains established until a call to `close()` terminates the endpoint. It also serves as a communication path if severe errors to the high speed network occur. In case of unmeant process termination through an illegal instruction, the `atexit()` function let *Sockets Direct* install handlers which are called prior to performing a final exit.

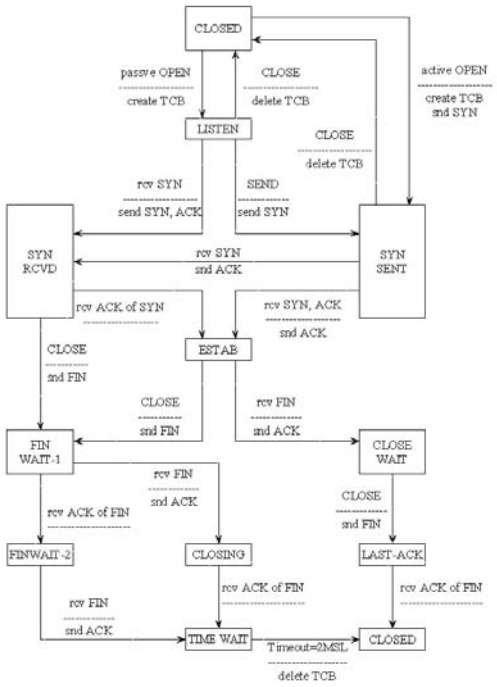

Figure 6.22: Socket TCP/IP States [48]

In the implementation several tokens are passed within a message header which serve as identifiers for receivers. This token can indicate an end of file (EOF) when closing a connection but also tokens for a shutdown of a socket exist. One important token is used for supporting a `fork()` operation. Since the call to fork is already a very expensive operation itself, a protocol to handle this event does not have the requirement to be very efficient. A much more anticipated task for this protocol is to let applications continue their operation.

The support of fork is very crucial for process management and many Internet services make heavy use of it.

In the following the problem coming with fork is described, it is depicted in figure 6.23.

When providing a sockets implementation which is running in user level mode, a dispatching instance such as the operating system is outside of the critical path. The problem arises when receiving data. Using the traditional TCP/IP stack, the OS was holding data until it was requested by a process

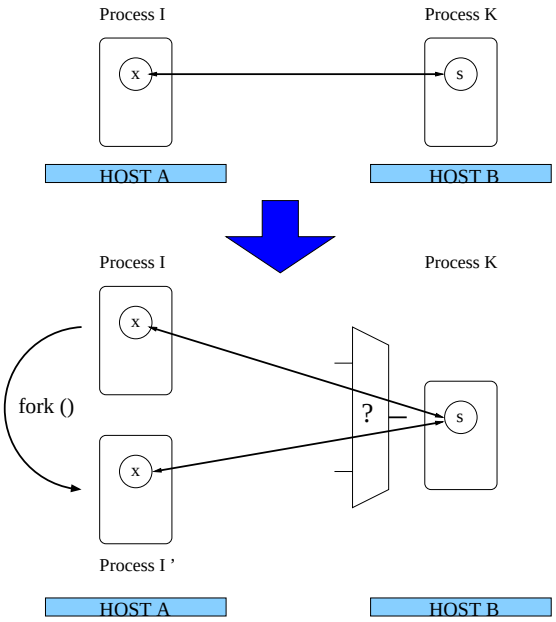

Figure 6.23: The fork() system call putting sockets in shared mode

using an assigned socket descriptor. When calling fork, all file descriptors including socket descriptors are shared between two or more processes. As a consequence the *Sockets Direct* implementation would receive data and put it into an address space in which it should not be copied. This way it is very hard to implement a solution for this scenario. Especially a platform independent approach is highly desirable. Some SAN's also do not support the shared usage of one port by more than one process. The solution which *Sockets Direct* does provide is achieving binary compatibility using the interception of the fork call itself. The rather complicated strategy uses tokens to inform communication partners about a fork command to be in action. These tokens are sent out before the original fork command is called. Furthermore, the initiating process waits for receivers to respond to this token. Several situations need to be handled.

An pseudo code implementation is presented in the following

```
pid_t fork(void)
{   // sending FORK_INFORM_SHARED_REQ on all fd's
    list = get_current_fd_using_SAN();
    for each socket i in list {
      set_wait_for_ack_on_socket (i);
      send (i, FORK_INFORM_SHARED_REQ);
      // adding sharing for sock i
      add_sharing_for_socket (i);
      // wait for fd ack so that we know remote process will
      // see socket to be shared
      waiting_for_shared_ack (list)
   }
   for each socket i in list {
      if (socket i holds message)
          send_back_message_to_origin ();
   }
   for each socket i in list {
      send_fork_inform_shared_finish (i);
   }
   // we have relayed messages back to their
   // origin, because we do not know whether
   // parent or child is going to receive them

   // call original fork function
   child_pid = SAN_TRAMP_fork();
   if (child_pid == 0) {
      // set up the child to get its own port
      san_startup_and_init();
      // connect to remote fd's
      for each socket i in list
           send (i, FORK_CONNECT_REQ);
      for each socket i in list
          recv (i, FORK_CONNECT_ACK);
   } else {
        // to prevent race conditions, the
        // parent also waits for ACKS
        for each i in list
            recv (i, FORK_CONNECT_ACK);
   }
   return child_pid;
}
```

Prior to calling fork, an endpoint may have sent data to the fork'ing process. In this case, the data is taken from the network and send back to its origin. As a portable solution we found that the concept of relying on the traditional network when sockets are shared is a viable approach. The origin which is aware of a shared socket will now use the traditional software stack and the parent and forked child will themselves issue calls to the traditional software stack. This mechanism however would slow down data exchanges for the remaining execution time. This behavior however can be reverted if one of the process is closing the shared socket. Fortunately, this is the case for most of the applications. As a consequence the communication partners are informed that a socket has been closed by one communication partner. A final message with updating tokens will bring the communication back to using the companion socket. The following pseudo source code will depict the establishment of a bijective point to point connection, after a shared socket has been closed by other communication partners:

```
token = get_msg_token (msg);
src = get_src (msg);
s = get_socket (msg);
switch (token) {
   ...
   CASE SOCK_SEND_EOF:
        // if the socket has been shared before,
// a point to point connection can be
// reused
        if (sharing has been deleted for s) &&
   if (s is no longer shared) &&
      if (connection partners for s = 1)
   send (SOCK_NO_LONGER_SHARED);
        else {
   // the socket was in non shared mode
   // before and the (only) communication
   // has closed the socket
   set_socket_to_have_rcvd_EOF (s);
   delete s from socket list used by SAN;

   // will indicate EOF to upper layers (recv)
   return msglen = 0;
       }
```

```
    ...
}
```

6.6 Efficiency of RDMA Enabled Data Transfers

This section focuses on providing insights on the performance discrepancy of high speed networks and host CPU and memory bandwidth. While networks become faster and faster (10 Gbit/s Ethernet implementation are available), the host CPU and its memory bandwidth experience less performance improvements [129]. This leads to starving networks and high host processing overhead when performing network I/O.

6.6.1 Motivation

The problem mentioned above is often referred to as the "I/O bottleneck" [18]. More specifically, the copying of data leads to high host overhead.

While TCP offload engines (TOE) lower the amount of time spent for checksum computation, the number of data movements is not addressed. Especially multiprocessor machines and clusters with high bandwidth feeds are affected from copying overhead. Such machines range from database servers, storage servers, application servers for transaction processing to clusters in scientific computing.

Clustered systems however establish only local connections for solving problems in parallel. Wide area network connections are only requested, when clusters are merged together to solve GRID applications.

Because of high end-host processing overhead in current implementations, the TCP/IP protocol stack is not used for high speed transfer. Instead special purpose network fabrics, system area networks like VIA [56], Quadrics [49], SCI and Myrinet [9], using remote direct memory access (RDMA) have been developed and are widely used. The I/O bottleneck has been an active field for research over the last years. The problem was addressed when high speed meant 100 Mbits/s FDDI and Fast Ethernet. With 10 Gbits/s Ethernet becoming available the research on protocol enhancements has gained popularity again [36], [37]. Moreover, user level protocols are now also used for file access enhancements [19]. Figures 5.1 and 5.2 present significant performance enhancements.

6.6.2 Reducing data relocations for Communication

The performance degradation of high speed networks has been confirmed in recent work [131], [130], [16]. Earlier work [17] pointed out operating system costs such as interrupts, context switches, process management, buffer management and timer management to be one reason for TCP overhead. Other factors come with processing individual bytes, specifically computing the checksum and moving data in memory. In this work, bandwidth was distinct to be the greatest source of limitation in which 64% of the measured microsecond overheads were caused by data touching operations.

Other work [14] highlighted operations such as data touching, checksum computation and memory copies to cause the most overhead for messages longer than 128 Bytes. For smaller sized messages, per packet overheads dominate [34], [16].

The percentage of overhead due to data-touching operations increases with packet size, since time spent on per-byte operations scales linearly with message size [34].

This was examined in detail in [63]. Using memory to memory TCP tests with varying MTU sizes, the percentage of total software costs attributable to per-byte operations were:

- 1500 Byte Ethernet 18-25%
- 4352 Byte FDDI 35-50%
- 9180 Byte ATM 55-65%

Although, many studies report results for data-touching operations including both checksumming and data movement together, much work has focused just on copying [14], [13], [55].

6.6.3 Limitations and Impact of Memory bandwidth

A number of studies show that eliminating copies substantially reduces overhead. For example, results from copy-avoidance in the IO-Lite system [43], which aimed at improving web server performance, show a throughput increase of 43% over an optimized web server, and 137% improvement over an Apache server.

There are many other examples where elimination of copying using a variety of different approaches showed significant improvement in system performance [15], [22], [26], [35], [55], [57].

Recent work by Chase et al. [16], measuring CPU utilization, shows that avoiding copies reduces CPU time spent on data access from 24% to 15% at 370 Mbits/s for a 32 KBytes MTU using a Compaq Professional Workstation and a Myrinet adapter [9]. This is an absolute improvement of 9% due to copy avoidance.

The total CPU utilization was 35%, with data access accounting for 24%. Thus the relative importance of reducing copies is 26%. At 370 Mbits/s, the system is not very heavily loaded. The relative improvement in achievable bandwidth is 34%.

When removing checksum computation, copy avoidance reduces CPU utilization from 26% to 10%. This is a 16% absolute reduction, a 61% relative reduction, and a 160% relative improvement in achievable bandwidth. This checksum computation is nowadays partly handled by high end networking cards, thus reducing another source of per-byte overhead. Another technique is to coalesce interrupts to reduce per-packet costs.

As a consequence, copying costs cause larger per centage of CPU utilization than before. Reducing data movements therefore is very promising for gaining increased performance, especially since the performance gap between host CPU and memory bandwidth is going to increase [129]. Faster CPU's also do not help in improving this situation [16].

Last but not least it is the application which will experience better performance for servicing clients for example. If databases can avoid spending time in network I/O, they will have more time for responding to their clients.

6.6.4 Using Remote direct memory access (RDMA) to Gain Performance Improvements

In early work, one goal of the software approaches was to show that TCP could go faster with appropriate OS support [17], [15].

Further investigation and experience showed that specific system optimizations have been complex, fragile, extremely interdependent with other system parameters in complex ways, and often of only marginal improvement [15], [16], [63], [21], [43].

For example, the Solaris Zero-Copy TCP work [63], which relies on page remapping, shows that the results are highly interdependent with other systems, such as the file system, and that the particular optimizations are specific for particular architectures, meaning for each variation in architecture optimizations must be re-crafted [63].

A number of research projects and industry products have been based on a memory-to-memory approach to copy avoidance. These include U-Net [26], SHRIMP [11], Infiniband [33] or Winsock Direct [47].

Several memory-to-memory systems have been widely used and have generally been found to be robust, to have good performance, and to be relatively simple to implement. These include all major system area networks. Networks based on these memory-to-memory architectures have been used widely in scientific applications and in data centers for block storage, file system access, and transaction processing.

The concept of using RDMA has attracted a large class of applications which takes advantage of memory to memory capabilities, including all the major databases, as well as file systems such as DAFS [19].

6.7 Optimizations

6.7.1 Analysis and Implementation of a Zero Copy Implementation

If the application is granted direct control over buffer management at the network level, it is natural to consider implementing a zero-copy interface to the protocol code. Unfortunately this optimization is awkward to implement with the BSD sockets API, since applications specify the area into which they wish received data to be delivered, requiring a copy to move the data to this location. Trapeze [58] attempts to improve on this by using page remapping to get buffers to user level, thus avoiding the copy. However, this technique has a number of drawbacks: changing page table entries can be quite expensive, particularly on multiprocessor machines, and there are security issues associated with receiving data payloads that are not an exact multiple of a page in size, since applications should be prevented from viewing old data belonging to others. The benefits of such *page flipping* to avoid the copy are often overstated, as the micro-benchmarks used to measure performance typically do not access the data being transferred. Thus the beneficial side-effect of the copy bringing the data into the processor's caches is not taken into account. In any case, it is generally necessary for the application to be written so that it uses page-aligned buffers and always issues reads with a length that is a multiple of the page size. Since most applications will need to be modified to satisfy these constraints, it may be little more effort to adapt the application to use a different API designed specifically

to enable zero-copy operation. Zero-copy APIs have been developed by a number of groups, but as yet no standard has emerged.

6.7.2 Protocol Threshold values for Efficient Communication

Sockets Direct offers three different protocols. A buffering protocol in which data is copied into a private buffer pool, thus achieving very low latency at application level, a zero copy strategy using *put* semantics and a zero copy strategy using *read* semantics.

Typically, two applications can agree on threshold values which are used to determine a specific protocol. The buffered copy is then used for smaller messages, while a rendezvous can be used for large messages. This threshold value can not be estimated in advance. Moreover, it is not possible to determine such a value in general. This value is dependent on the application. For micro benchmarks, which are only sending and receiving data, but do not perform any additional computation, a rendezvous protocol for CPU offloading only achieves good performance for very large messages. For a real world application which does additional operations such as file access or number crunching, the achieved bandwidth is not the most important factor. It is the reduced CPU load which will improve the overall throughput of the system.

6.7.3 Enhancing Data Copies

One aspect in terms of enhancements would be the avoidance of data copies as it was described in [55]. *Sockets Direct* as a thin independent layer was designed and implemented to avoid unnecessary data copies under any circumstances. This is not the case if unexpected messages arrive which are taken from the network after a timeout value. In this case, the data will be copied into newly allocated applications memory. When this data is later on requested, the allocated memory will be freed. It is important to know that the functions `malloc()` and `free()` are optimized in such a way, that the `free()` command does not give back allocated memory back to the operating system but will keep it for a following `malloc()`. This way, there is only one penalty in terms of performance when allocating memory for the first time.

Under *Sockets Direct*, when using a buffered protocol, it is quite clear, that the communication performance is very much dependent on the memory performance of the host system. In this case data is copied into a private

buffer pool, before DMA engines of a NIC will spool data from the host interface into the network interface.

When designing and implementing *Sockets Direct*, a request for 64bit aligned memory chunks will lead to better performance of `memcpy` operations since CPU specific and optimized instructions can be used. This performance is also very dependent on the chipset being used. Performance measurements were conducted on a variety of systems. These were positioned in the low cost area. The difference of performance for a micro benchmark which was simply measuring the memory performance and the consequence for a communication layer, are depicted in Figure 6.24. The results are identifying that memory bandwidth is the cause for lower bandwidth. Systems having faster memory interfaces will be able to pass this performance to the communication subsystem. The `membench` [73] micro benchmark will provide performance information when copying data from one memory region to another. It will measure the time needed for copying the data and results are given in MBytes/s. Typically one can distinguish easily the L1, L2, L3 cache sizes. The achieved bandwidth for very long data transfers will then show the actual performance of the memory system. On the other hand, the netperf benchmark will conduct a test in which data is exchanged between two hosts over a network. It will run as a streaming test in which one host continously sends data to another.

6.7.4 Additional Interception for Data Compression or Encryption

The interception technique implemented for *Sockets Direct* could also be used to perform other tasks. Currently it routes payload to the destination using a high speed network. However it could also be applied to modify the payload being send. One aspect could be data compression, given the very fast and powerful CPU's of today. Here, slower networks could experience higher throughput since the data is compressed at the source and decompressed at the destination. This would result in higher bandwidth since less data is on the wire. This is similar to the Nagle algorithm implemented in the TCP stack which accumulates smaller packets. An interception of MPI calls to transfer a picture between two nodes resulted in higher throughput using compression. This technique however can not be applied in general, as other tests for synthetic data introduced additional overhead when using compression. A different application however is that of data security. A middleware layer could implement data encryption to transfer sensitive data. An

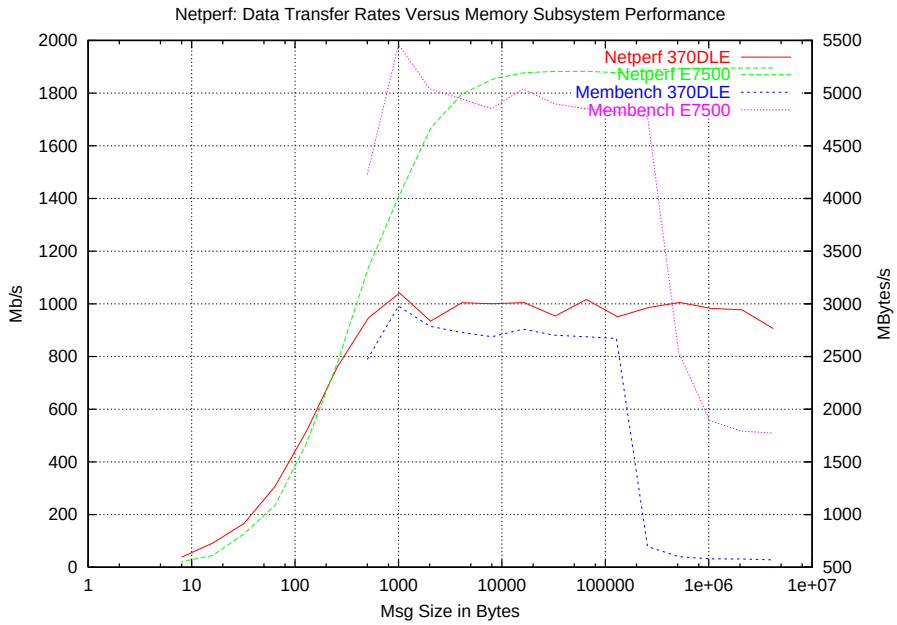

Figure 6.24: Comparison of Netperf Performance and Membench Performance using High End SDRAM/DDR-RAM Systems

alternative to the Secure Sockets Layer (SSL) which requires a source code modification.

6.7.5 Connection Establishment

The current implementation also offers room for further improvement. As such the concept on how connections are established can be targeted. When creating a companion socket, the `accept()` and `connect()` function calls are intercepted and added with new functionality. Still, original functions are called and the convenience coming with this implementation is that the operating system will fill in connection information. If an application should only establish a connection for a very short amount of time than significant overhead comes with the traditional concept. Moreover, the exchange of required information about the system area network using standard communication will have the communication start up costs of TCP. Instead a resource broker could be implemented as a central instance. This broker could be queried to gather necessary information about server or clients, respectively. When avoiding traditional accept/connect pairs, and a subsequent data exchange

using traditional send and receives, the *Sockets Direct* approach would be able to increase the number of very short transactions.

6.8 Performance Analysis

In this section we would like to give a comparison on using existing protocols and different high speed networks.

6.8.1 TCP / IP over System Area Networks

6.8.1.1 Tuning TCP/IP over System Area Networks

The Transmission Control Protocol contains a vast of parameters. Most of them can be neglected in terms of improving the communication performance. In this section, the parameters are addressed which have the most influence on performance.

- MTU
- Socket Buffer Sizes
- TcpWindowSize

The TCP receive window size is the amount of receive data (in bytes) that can be buffered at one time on a connection. The sending host can send only that amount of data before waiting for an acknowledgment and window update from the receiving host. In Windows 2000, the TCP/IP stack was designed to tune itself in most environments and uses larger default window sizes than earlier versions. Instead of using a hard-coded default receive window size, TCP adjusts to even increments of the maximum segment size (MSS) negotiated during connection setup. Matching the receive window to even increments of the MSS increases the percentage of full-sized TCP segments used during bulk data transmission. There are two methods for setting the receive window size to specific values:

The TcpWindowSize registry parameter and the `setsockopt` Socket function (on a per-socket basis). To improve performance on high-bandwidth, high-delay networks, scalable windows support (RFC 1323) has been introduced in Windows 2000. This RFC details a method for supporting scalable windows by allowing TCP to negotiate a scaling factor for the window size

at connection establishment. This allows for an actual receive window of up to 1 GigaByte (GB). RFC 1323 Section 2.2 provides a good description [29].

Another way for optimizing the TCP/IP performance is the tuning of the MTU.

Path Maximum Transmission Unit (PMTU) Discovery:
PMTU discovery is described in RFC 1191. When a connection is established, the two hosts involved exchange their TCP maximum segment size (MSS) values. The smaller of the two MSS values is used for the connection. Historically, the MSS for a host has been the MTU at the link layer minus 40 bytes for the IP and TCP headers. However, support for additional TCP options, such as time stamps, has increased the typical TCP+IP header to 52 or more bytes.

The PMTU between two hosts can be discovered manually using the ping command with the -f (don't fragment) switch, as follows:

```
ping -f -n <number of pings> -l <size> <destination ip address>
```

Slow Start Algorithm and Congestion Avoidance:

When a connection is established, TCP starts slowly at first to assess the bandwidth of the connection, and to avoid overflowing the receiving host or any other devices or routers in the path. The send window is set to two TCP segments, and if that is acknowledged, it is incremented to three segments. If those are acknowledged, it is incremented again, and so on until the amount of data being sent per burst reaches the size of the receive window on the remote host. At that point, the slow start algorithm is no longer in use, and flow control is governed by the receive window. However, congestion could still occur on a connection at any time during transmission. If this happens (evidenced by the need to retransmit), a congestion-avoidance algorithm is used to reduce the send window size temporarily and to grow it back towards the receive window size. Slow start and congestion avoidance are discussed further in RFC 1122 and RFC 2581 [30], [31].

Throughput Considerations:

TCP was designed to provide optimum performance over varying link conditions, and Windows 2000 contains improvements such as those supporting RFC 1323. Actual throughput for a link depends on a number of variables, but the most important factors are:

- Link speed (bits-per-second that can be transmitted)
- Propagation delay
- Window size (amount of unacknowledged data that may be outstanding on a TCP connection)
- Link reliability
- Network and intermediate device congestion
- Path MTU

TCP throughput calculation is discussed in detail in Chapters 20-24 of 'TCP/IP Illustrated', by W. Richard Stevens [135]. Some key considerations are listed below: The capacity of a pipe is bandwidth multiplied by round-trip time. This is known as the bandwidth-delay product. If the link is reliable, for best performance the window size should be greater than or equal to the capacity of the pipe so that the sending stack can fill it. The largest window size that can be specified, due to its 16-bit field in the TCP header, is 65535, but larger windows can be negotiated by using window scaling as described earlier in this document. Throughput can never exceed window size divided by round-trip time if the link is unreliable or badly congested and packets are being dropped, using a larger window size may not improve throughput. Along with scaling windows support, Windows 2000 supports Selective Acknowledgments (SACK; described in RFC 2018) to improve performance in environments that are experiencing packet loss. It also includes support for timestamps (described in RFC 1323) for improved RTT estimation.

6.8.2 Performance of Micro Benchmarks

In this section the performance of several well known micro benchmarks will be presented. Given the eliminated overhead for *Sockets Direct*, latency should lower significantly and bandwidth should be improved close to the available performance from the low level API of the interconnect. For conducting these tests, systems have been used which offer very good PCI performance. As an interconnect Myrinet [9] was chosen. The system consisted of 2 * PIII 1Ghz plugged into a Supermicro Board using the Serverworks chipset. The PCI bus performance, having a 64bit/66Mhz interface, was measured to be 478 MBytes/s bus read and 512MBytes/s bus write. The

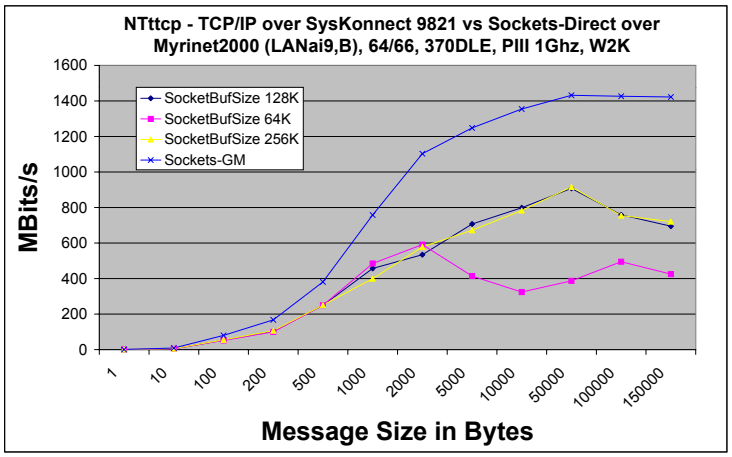

Figure 6.25: NTttcp Performance

low level API performance using GM over Myrinet was 9usec latency and 243MBytes/s for bandwidth.

Figure 6.25 depicts the result when running the NTttcp benchmark. This benchmark based on the ttcp benchmark has been improved for the Windows Operating System. It offers a comparison for two different interconnect types, Gigabit Ethernet using a high end network card from Syskonnect and the Myrinet 2000 network card from Myricom. The tests were run using a tuned TCP/IP implementation over Syskonnect and a *Sockets Direct* implementation using GM over Myrinet. It may be noted that the network interfaces were plugged into the same host system. Thus the tests reflect a fair comparison since the used hardware components are identical. Only the operating system has been tuned to maximize the performance. When looking at the graphs the low latency for *Sockets Direct*, which have been measured to be only 2usec higher than the low level API for Myrinet, results in much better performance beginning with small messages. The graphs also show the Syskonnect performance using several TcpWindowSizes (see also section 6.8.1.1). It becomes clear how important the correct TcpWindowSize for larger messages becomes. If the wrong value is chosen (in this case it was the default value), the performance degrades significantly (up to 50%).

But also the socket buffer sizes have been adjusted to get optimal performance. As stated before, the Syskonnect Gigabit Ethernet card is a high end network interface. In a large comparison with other Gigabit Ethernet cards, it has distinguished itself from other vendors [78].

Another popular benchmark is the Iperf benchmark [79]. It is a modern tool to measure network performance. Similar to the results above the *Sockets*

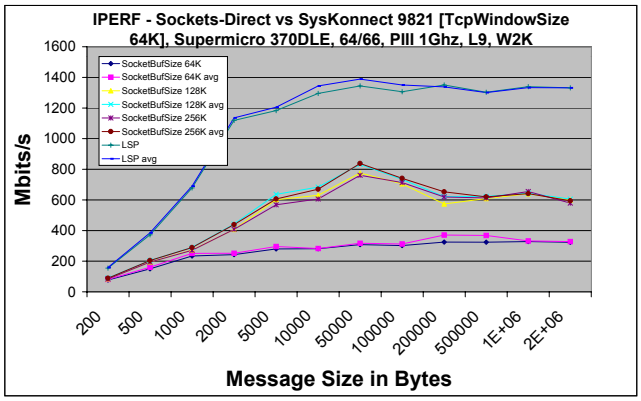

Figure 6.26: IPerf Performance

Direct Layer increases the performance and outperforms Gigabit Ethernet significantly. The following figure depicts the performance result from a conducted test using netpipe [77]. This time, the performance graphs also include the results for a TCP/IP over Myrinet layer.

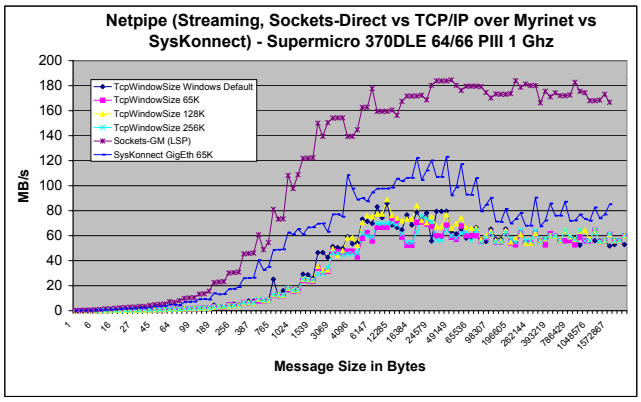

Figure 6.27: Netpipe Streaming Performance

One remarkable aspect is that the TCP/IP stack used for Myrinet is less efficient than the TCP/IP stack over Syskonnect. Still, the *Sockets Direct* implementation outperforms any alternative strategy. It offers superior performance for small messages with twice the performance and scales very well for larger messages as well. For larger messages, the performance of Syskonnect Gigabit Ethernet drops significantly. When conducting this test, the TCP/IP over GM performance peaks at 84MBytes/s, the Syskonnect Gigabit Ethernet performance peaks at 120MBytes/s, both TCP/IP imple-

mentations showing high variances. The *Sockets Direct* implementation using GM over Myrinet peaks at 191MBytes/s.

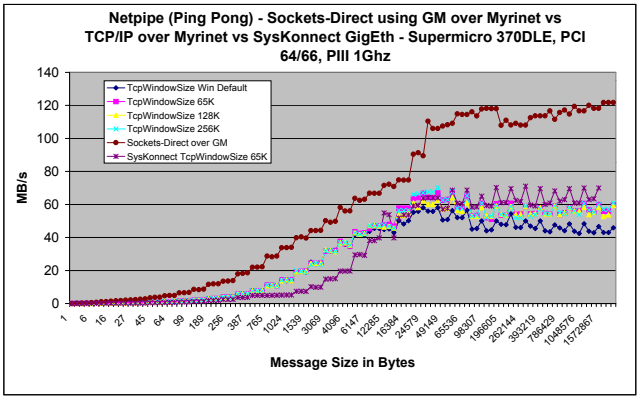

Figure 6.28: Netpipe Round Trip Performance

Sockets Direct does not only bypass TCP function calls, but has been extended to handle UDP operations as well. The following figure depicts the performance gain using UDP. UDP in general is unreliable. The *Sockets Direct* however relies on reliable low level API calls making the UDP protocol reliable.

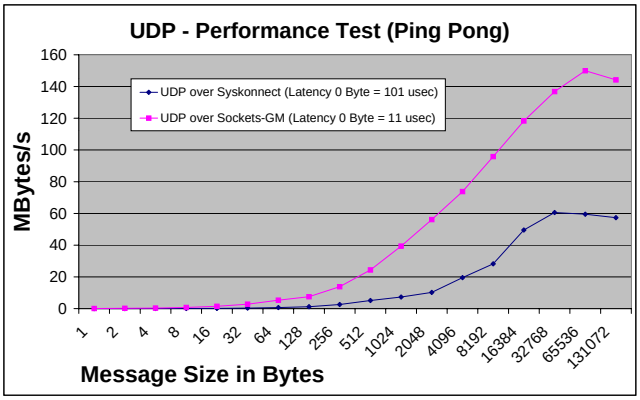

Figure 6.29: UDP Performance

6.8.3 Host CPU Utilization Measurements

Transferring data with low latency and high bandwidth are one requirement for increasing applications performance. Another factor with almost the same

importance is the overhead associated with message transfers. Basically two methods exist in todays systems. Given that the cluster market is targeting low cost systems, the interconnection provider has to rely on available interfaces. The design space for interconnection networks as described in 2.2 is thus very limited for a practical approach. In principal, message transfer can be performed by using Programmed Input Output (PIO) or DMA capable network interfaces. PIO is typically used for very small messages

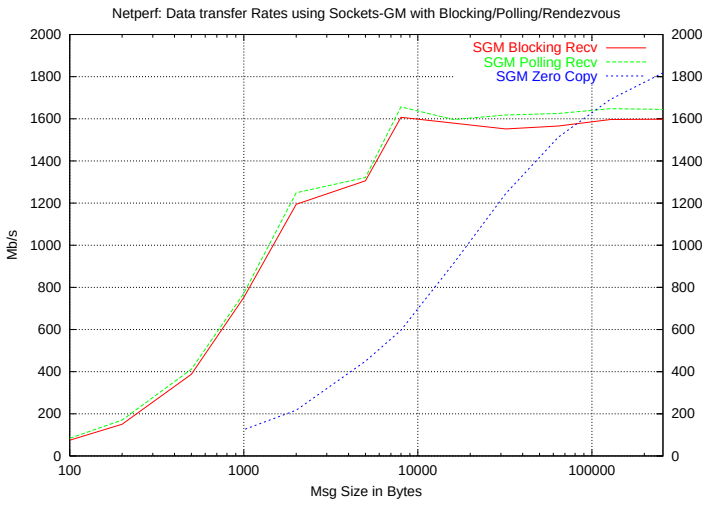

Figure 6.30: Netperf Performance Using Blocking, Polling and Rendezvous Strategies

and involves the CPU which directly injects messages into the network. For larger messages, PIO can also still be used, however, the application will wait for the CPU to finish the transfer. When using DMA , a descriptor will be assembled, describing the data to be transfered. This includes the physical address from which the data can be fetched as well as the data length. There are still other parameters which need to be addressed. A more detailed description can be found in chapter 5. However, as a consequence, the CPU is no longer involved in message transfers but can do other computation. In the following we will present performance results which in addition of network measurements also analyze the involved CPU overhead. The test conducted examined the performance of three available strategies for sending and receiving messages. When using blocking, then the receiving application will wait to be triggered by the network interface when data is

available. Polling will check the message queue and rendezvous will exchange information for sending direct messages which will be stored directly in the final message buffer. There are two conclusions to be taken from the figure above. One remarkable observation is that there is almost no difference when using blocking or polling receives. Usually interrupts for triggering a process are considered to be costly. The performance of a rendezvous strategy is poor for small messages, but becomes better for larger messages. Before drawing a conclusion, the CPU load involved when using different strategies is examined. Figure 6.31 depicts the performance of *Sockets Direct* using

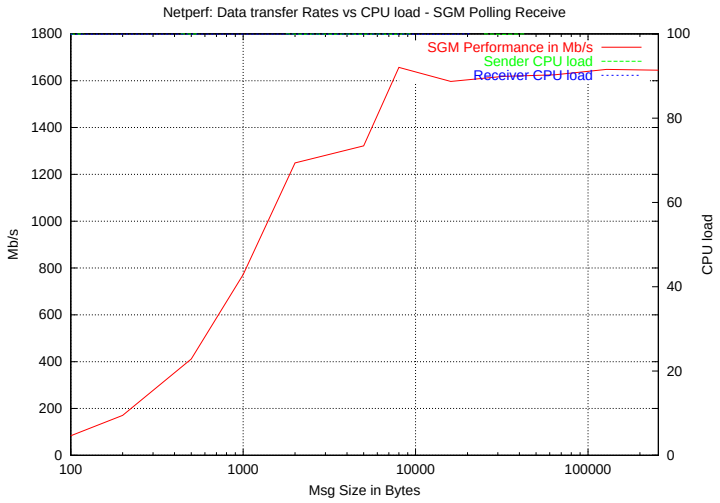

Figure 6.31: Netperf Performance versus CPU load Using Polling Receive

GM over Myrinet in comparison with the CPU load. The `gm_receive()` function will poll the message queue until data has arrived. This results in a CPU load of 99.9%, which can be found in the diagram above to be matching the 100% line. This will undoubtly have a negative effect on other applications running on the same system. Figure 6.32 depicts the *Sockets Direct* performance in relation with CPU overhead using blocking receive. `gm_blocking_receive_no_spin()` does not poll the message queue but will be interrupted by the NIC. Surprisingly, the gained performance is almost the same, but CPU load is much lower, averaging in a CPU load of 55%. The *Sockets Direct* performance using a zero copy strategy offers acceptable performance for a micro benchmark only for large messages. Beginning with a message size which is large enough to overcome the overhead issued with the rendezvous protocol, the *Sockets Direct* performance however is better

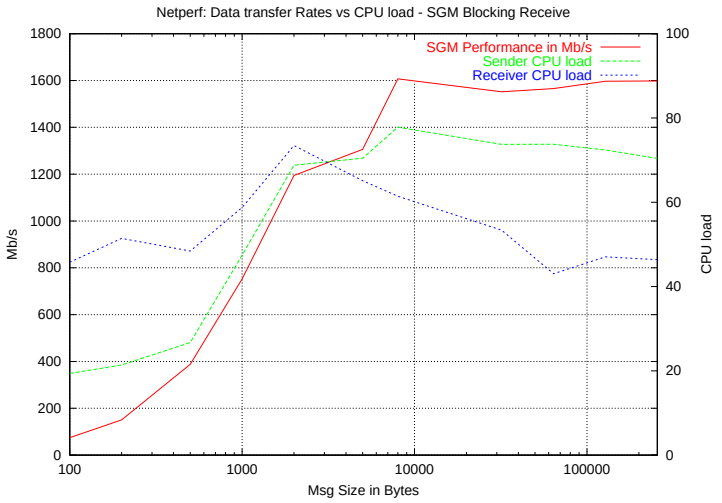

Figure 6.32: Netperf Performance versus CPU load Using Blocking Receive

than buffered copies. One major reason is the performance of the memory subsystem. Still, transferring data with 1.8Gbit/s with a CPU load of only 3% is remarkable. As explained earlier the effect of these performance results may vary from application to application.

6.9 Sockets Direct Enhancements to Legacy Applications

System Area Networks are currently being considered a special type of network which requires special communication layers such as MPI or PVM to speed up applications. For this, applications need to be redesigned and implemented again. For quite a few applications this is a non feasible approach. While some applications like a distributed database have added support for VIA, this interface will no longer be in favor [111]. Instead emerging protocols like the direct access transport (DAT) specification [112] which overcome several of VIA's drawbacks will be provided. The socket interface which will be driving the Internet over the next years however is an existing interface for distributed applications. Hence, these applications will experience a performance gain without involving additional development from the application developer. Finally, this would broaden the usage of System Area Networks

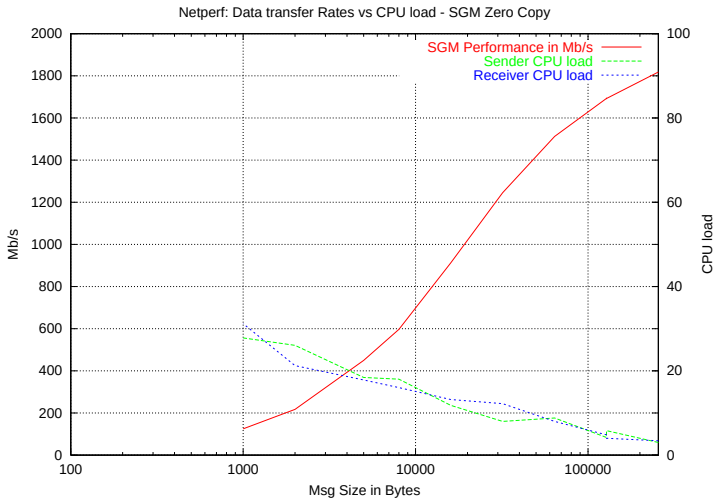

Figure 6.33: Netperf Performance versus CPU load Using Rendezvous

and open different categories of usage. This would be due to the fact of binary compatibility level of applications using System Area Networks.

6.9.1 Increasing Transaction Numbers For Databases

One currently growing market is that of e-commerce. B2B and B2C platforms have become more and more important with the Internet boom over the recent years. The back end of all these platforms are databases. In this scenario, the WWW clients which request their information will access or query application servers which then query the database server. The following figure depicts an overview of such a scenario. In this case, the connection from WWW clients to the application servers can not be improved and is depending on other traffic from the Internet as well. However, the connection from application servers to the database server could be improved significantly given that a *Sockets Direct* offers an order of magnitude on communication performance with respect to latency and bandwidth. This improvement is gained when looking at micro benchmarks like `netpipe` or `netperf` which have been presented above (see section 6.8). However, it is not clear which performance gain a real world application will have when the communication performance improves significantly. When looking at the overall performance using various connection types, for some applications the benefit of a high speed interconnect is not visible. Thus a large fraction of clusters are still

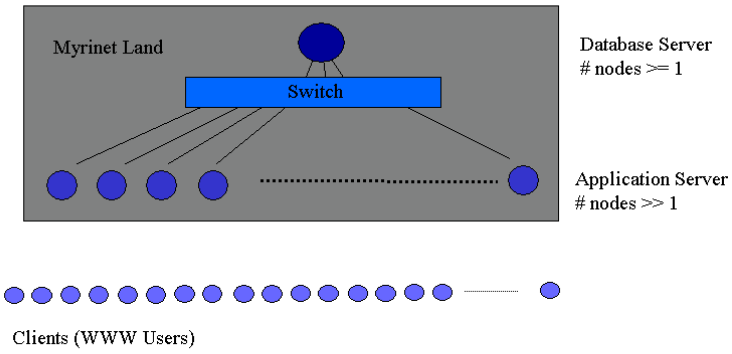

Figure 6.34: Database Server, Application Servers and Clients

equipped with Fast Ethernet since it basically comes for free [74]. As of today, a high speed interconnect such as Myrinet or SCI will double the cost for a compute node, assuming a low cost system. Therefore we have chosen a real world application which is in intensive usage. In the following we present the results of a load test which has been conducted between an application server and a database server according to 6.34. This load test started 50 queries to the database.

	Min T.	Max T.	Avg T.	Tot.T.	ODBC T.	Exec T.	Fetch T.
TCP/IP	62	969	92	4531	4624	4189	373
Sockets-GM	62	79	71	3505	3520	3400	152

As a result, the performance improvement is indeed encouraging. The Avg Time has been improved by 35%. Moreover, the Total Time has been minimized by 23%. The Fetch Time has been decreased by 60%. These results show that the *Sockets Direct* Protocol is very efficient as well for real world applications like a database. Using the Total Time for specifying the number of transactions, then they can be improved by more than 28%.

6.9.2 Distributed Applications

Besides the example of a database which is queried by application servers, there are several other applications which could benefit from a efficient communication protocol like *Sockets Direct*. For example, the IBM DB2 is a decentralized database which can be seen as a parallel application, the sockets

interface is used for communication. Other application are DCOM applications or Corba applications.

6.10 Sockets Direct for the ATOLL Network

In this section a description will be given in which way, the *Sockets Direct* concept which has been fully implemented using the Myrinet network can be applied to other SANs such as the ATOLL network.

An important requirement is that the low level API of the SAN guarantees the correct delivery of the message or reports a failure. For this, it is also assumed, that error detection and correction will be handled either by the hardware itself, or the low level API. For the ATOLL network, these error handling is supported by its hardware and thus the network allows for a full implementation.

6.10.1 Mapping of Sockets Direct functions

The ATOLL API which is using message passing style primitives itself when exchanging data makes the porting of code easy. Without pre-registered buffers, which have to be released in order to guarantee safe delivery of messages and to avoid data corruption by copying data into buffers before the transmission has ended, some performance will be gained, too.

The *Sockets Direct* adds a small header of two integers to identify a message correctly. This is because a process which holds one port may open several connections to another process and thus another identifier is needed to lookup the corresponding remote descriptor. When sending a message, then the migration is as simple as replacing GM send function calls with ATOLL send function calls. The lookup for an appropriate buffer under GM can be avoided. When receiving, the ATOLL API will directly store incoming data into the provided buffer. Thus, a direct mapping can be achieved, too.

6.10.2 Shared Socket Handling

When sockets are shared, then the ATOLL API experiences the same difficulties like other SANs which operate on a point to point bases.

For this, the *Sockets Direct* for ATOLL implementation must also intercept the `fork()` function call which would otherwise allow two or more

separate processes access a single ATOLL hostport. This is not supported and the software to work around this problem would involve high cost.

That way, the new `fork()` routine will first let the client process set the ATOLL handle to be invalid. Then, a new ATOLL hostport can be opened.

6.11 Conclusion

This chapter was describing the design and implementation of a new middleware layer which replaces traditional, overburdened protocols which are used for almost any distributed application.

This middleware layer achieves binary compatibility which revolutionizes the usage of system area networks. With this highest level of compatibility, legacy applications will work right out of the box. Moreover, the usage of system area networks experiences a much broader diversification. Thus, system area networks can now be seen as regular networks.

Other very important results are that the raw bandwidth of a system area network can be given to the application. The throughput of the overall host system will also be enhanced by using RDMA buffer semantics.

The performance improvements have been verified by a set of objective micro benchmarks. The results show that a performance improvement in the order of a magnitude by enabling the middleware layer.

But also the question has been solved, in which way real world application can benefit from faster networks and efficient middleware layers. For this, the throughput of a SQL database has been improved by 28%, a result which motivates further investigations into measuring other database benchmarks.

When looking at the design space, this innovative middleware layer also addresses its usage and applicability on different operating systems.

The implementation is portable, it is not restricted to a single specific system area network.

Bibliography

[1] D. Culler et al. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, San Diego, CA, May 1993.

[2] R. P. Martin, A. M. Vahdat, D. E. Culler and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pp. 85-97, June 1997.

[3] D. Cheriton and C. Williamson. VMTP: A transport layer for high-performance distributed computing. *In IEEE Communications*, pp. 37-44, June 1989.

[4] R. M. Watson and S. A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *In ACM Transactions on Computer Systems*, 5(2):97-120, May 1987.

[5] K. Keeton, D. A. Patterson, and T. E. Anderson. LogP quantified: The case for low-overhead local area networks. *In Hot In-terconnects III*, Stanford University, Stanford, CA, August 1995.

[6] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *In Proceedings of SIGCOMM '90*, pp. 200-208, Philadelphia, Pennsylvania, September 1990.

[7] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is layering harmful? *IEEE Network*, 6(1):20-24, January 1992.

[8] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *In IEEE/ACM Transactions on Networking*, 1(5):600-610, October 1993.

[9] N. J. Boden, D. Cohen, 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*, February 1995.

[10] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, J. Wilkes. An implementation of the Hamlyn send-managed interface architecture. *in Proceedings of the Second Symposium on Operating Systems Design and Implementation*, USENIX Assoc., Oct. 1996.

[11] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki and E. W. Felten. A virtual memory mapped network interface for the SHRIMP multicomputer. *in Proceedings of the 21st Annual Symposium on Computer Architecture*, April 1994, pp. 142-153.

[12] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. D. Kubiatowicz. Remote Queues: Exposing message queues for optimization and atomicity. *In Proceedings of SPAA '95*, Santa Barbara, CA, June 1995.

[13] J. C. Brustoloni. Interoperation of copy avoidance in network and file I/O. *In Proceedings of IEEE Infocom*, 1999, pp. 534-542.

[14] J. C. Brustoloni, P. Steenkiste. Effects of buffering semantics on I/O performance. *in Proceedings OSDI'96, USENIX*, Seattle, WA Oct. 1996, pp. 277-291.

[15] C-H Chang, D. Flower, J. Forecast, H. Gray, B. Hawe, A. Nadkarni, K. K. Ramakrishnan, U. Shikarpur, K. Wilde, High-performance TCP/IP and UDP/IP networking in DEC OSF/1 for Alpha AXP. *In Proceedings of the 3rd IEEE Symposium on High Performance Distributed Computing*, August 1994, pp. 36-42.

[16] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End system optimizations for high-speed TCP. *In IEEE Communications Magazine , Volume: 39, Issue: 4*, April 2001, pp 68-74. http://www.cs.duke.edu/ari/publications/end-system.pdf

[17] D. D. Clark, V. Jacobson, J. Romkey, H. Salwe. An analysis of TCP processing overhead. *In IEEE Communications Magazine*, Volume: 27, Issue: 6, June 1989, pp 23-29.

[18] D. D. Clark, D. Tennenhouse. Architectural considerations for a new generation of protocols. *in Proceedings of the ACM SIGCOMM Conference*, 1990.

[19] Direct Access File System http://www.dafscollaborative.org http://www.ietf.org/internet-drafts/draft-wittle-dafs-00.txt

[20] S. N. Damianakis, C. Dubnicki, and E. W. Felten. Stream sockets on SHRIMP. *Technical Report TR-513-96*, Princeton University, Princeton, NJ, October 1996.

[21] P. Druschel, M. B. Abbott, M. A. Pagels, L. L. Peterson. Network subsystem design. *in IEEE Network*, July 1993, pp. 8-17.

[22] P. Druschel, L. L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. *in Proceedings of the 14th ACM symposium of Operating Systems Principles*, Dec. 1993.

[23] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. Network subsystem design: A case for an integrated data path. *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4):8-17, July 1993.

[24] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adapter: A software perspective. *In Proceedings of ACM SIGCOMM '94*, August 1994.

[25] A. Edwards and S. Muir. Experiences in implementing a high performance TCP in user-space. *In ACM SIGCOMM '95*, Cambridge, MA, August 1995.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. *in Proc. of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, Dec. 3-6, 1995.

[27] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early experience with message-passing on the SHRIMP multicomputer. *In Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*, pp. 296-307, Philadelphia, PA, May 1996.

[28] R. Fielding, J. Gettys, J. Mogul, F. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. RFC 2616, June 1999.

[29] V. Jacobson, R. Braden and D. Borman. TCP Extensions for High Performance. Request for Comments: 1323, May 1992.

[30] R. Braden. Requirements for Internet Hosts - Communication Layers. Request for Comments: 1122, October 1989.

[31] M. Allman, V. Paxson and W. Stevens. TCP Congestion Control. Request for Comments: 2581, April 1999.

[32] Fibre Channel Standard. http://www.fibrechannel.com/technology/index.master.html

[33] InfiniBand Architecture Specification, Volumes 1 and 2, Release 1.0.a. http://www.infinibandta.org

[34] J. Kay, J. Pasquale. Profiling and reducing processing overheads in TCP/IP. *in IEEE/ACM Transactions on Networking, Vol 4, No. 6*, pp.817-828, Dec. 1996.

[35] K. Kleinpaste, P. Steenkiste and B. Zill. Software support for outboard buffering and checksumming. *In SIGCOMM'95*.

[36] K. Magoutis. Design and Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD. *in Proceedings of USENIX BSDCon 2002 Conference*, San Franscisco, CA, February 11-14, 2002.

[37] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, D. Gallatin, R. Kisley, R. Wickremesinghe and E. Gabber. Structure and Performance of the Direct Access File System (DAFS). *in 2002 USENIX Annual Technical Conference*, Monterey, CA, June 9-14, 2002.

[38] J. D. McCalpin. A Survey of memory bandwidth and machine balance in current high performance computers. *in IEEE TCCA Newsletter*, December 1995.

[39] C. Maeda and B. N. Bershad. Networking performance for microkernels. *In Proceedings of the Third Workshop on Workstation Operating Systems*, pp. 154-159, 1992.

[40] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. *In Proceedings of the Fourteenth Symposium on Operating Systems Principles* , pp. 244-255, Asheville, NC, December 1993.

[41] A. Newman. IDC report paints conflicted picture of server market circa 2004. *In ServerWatch*, July 24, 2000. http://serverwatch.internet.com/news/2000_07_24_a.html

[42] M. Pastore. Server shipments for 2000 surpass those in 1999. *in ServerWatch*, Feb. 7, 2001 http://serverwatch.internet.com/news/2001_02_07_a.html

[43] V. S. Pai, P. Druschel, W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *Proc. of the 3rd Symposium on Operating Systems Design and Implementation* , New Orleans, LA, Feb. 1999.

[44] S. Pakin, M. Lauria, and A. Chien. High-performance messaging on workstations: Illinois Fast Messages(FM) for Myrinet. *In Supercomputing '95*, San Diego, CA, 1995.

[45] L. L. Peterson. Life on the OS/network boundary. *in Operating Systems Review*, 27(2):94-98, April 1993.

[46] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas and K. Yelick. A case for intelligient RAM: IRAM. *in IEEE Micro*, April 1997.

[47] J. Pinkerton. Winsock Direct: the value of System Area Networks. http://www.microsoft.com/windows2000/techinfo/howitworks/communications/winsock.asp

[48] Postel, J. Transmission Control Protocol - DARPA Internet Program Protocol Specification. RFC 793, September 1981.

[49] Quadrix Solutions. http://www.quadrix.com

[50] Sockets Direct Protocol. Infiniband TA, 2002.

[51] C. Seitz. Myrinet: A gigabit-per-second local area network. *In Hot Interconnects II*, Stanford University, Stanford, CA, August 1994.

[52] Compaq Servernet. http://nonstop.compaq.com/view.asp?PAGE=ServerNet

[53] The STREAM Benchmark Reference Information, http://www.cs.virginia.edu/stream/

[54] C. A. Thekkath, T. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user-level. *in IEEE/ACM Transactions on Networking*, pp. 554-565, October 1993.

[55] M. N. Thadani, Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. *in Technical Report*, SMLI TR-95-39, May 1995.

[56] Virtual Interface Architecture Specification Version 1.0. http://www.viarch.org/html/collateral/san_10.pdf

[57] J. R. Walsh. DART: Fast application-level networking via data-copy avoidance. *in IEEE Network, July/August 1997*, pp.28-38.

[58] A. Gallatin, J. Chase, and K. Yochum. Trapeze/IP: TCP/IP at near-gigabit speeds. *in Proceedings of the USENIX 99 Technical Conference*, June 1999.

[59] A. Edwards and S. Muir. Experiences implementing a high performance TCP in user-space. *in Proceedings of ACM SIGCOMM 95*, September 1995, pp. 196–205.

[60] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrated communication and computation. *In Proceedings of the Nineteenth ISCA*, Gold Coast, Australia, May 1992.

[61] 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 the Fifteenth SOSP*, pp. 40-53, Copper Mountain, CO, December 1995.

[62] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. *In Proceedings the Proceedings of the International Parallel Processing Symposium*, pp. 388-396, 1997.

[63] J. Chu. Zero-copy TCP in Solaris. *In Proceedings of the 1996 Usenix Technical Conference*, pages 253–64, San Diego, CA, USA, Jan. 1996

[64] Loic Prylli. Linux OS issues with Myrinet: the good, the bad, and the ugly. *Second Myrinet Users Group Meeting*, Vienna, Austria, 2002

[65] G. Hunt and D. Brubacher Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[66] D. Clark. Window and acknowledgement strategy in TCP. Internet RFC 813, July 1982.

[67] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. em In SIGCOMM Annual Technical Conference '93, 1993.

[68] C. Maeda and B. Bershad. Service without servers. *In Workshop on Workstation Operating Systems IV*, October 1993.

[69] S. H. Rodrigues, Th. E. Anderson and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. *In Proceedings of the USENIX Annual Technical Conference*, January 6-10, 1997, Anaheim, California, USA.

[70] J. Kim, K.Kim, and S. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface. *In Proceedings of the IEEE Cluster 2001 Conference*, Newport Beach, CA, USA, 2001.

[71] http://oss.hitachi.co.jp/crl/lwsockets-en.html

[72] Shinji Sumimoto. A Study of High Performance Communication for Parallel Computers Using a Commodity Network. *PhD thesis*, Keio University, 2000.

[73] Ch. Kurmann, T. Stricker. Characterizing memory system performance for local and remote accesses in high end SMPs, low end SMPs and clusters of SMPs. *In 7th Workshop on Scalable Memory Multiprocessors*, held in conjunction with ISCA98, June 27-28 ,1998, Barcelona, Spain.

[74] http://clusters.top500.org

[75] http://www.netperf.org/netperf/NetperfPage.html

[76] J.J. Dongarra, H.W. Meuer and E. Strohmaier. Top500 List. http://www.top500.org

[77] Q. O. Snell, A. R. Mikler and J. L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. *In IASTED International Conference on Intelligent Information Management and Systems*, June 1996

[78] A. Betz and P. Gray. Gigabit Over Copper Evaluation. University of Northern Iowa, Technical Report TR040202-CS. http://www.cs.uni.edu/ gray/gig-over-copper/

[79] A. Tirumala, F. Qin, J. Dugan and J. Ferguson. http://dast.nlanr.net/Projects/Iperf/

[80] T. Curry. Profiling and Tracing Dynamic Library Usage Via Interposition. *In Proceedings of the USENIX Summer Technical Conference*, 1994.

[81] E. Speight, H. Shafi and J. K. Bennett. WSDLite: A Lightweight Alternative to Windows Sockets Direct Path. *In Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, WA, August 2000.

[82] U. Brüning and L. Schaelicke. Atoll: A High-Performance Communication Device for Parallel Systems. *In Proceedings of the 1997 Conference on Advances in Parallel and Distributed Computing*, IEEE CS Press, Los Alamitos, Calif., 1997, pp 228-234.

[83] L. Rzymianowicz, U. Brüning, J. Kluge, P. Schulz and M. Waack. ATOLL: A Network on a Chip. *In Cluster Computing Technical Session (CC-TEA) of the PDPTA'99 conference*, June 28 - July 1 1999, in Las Vegas, NV.

[84] U. Brüning. Lecture Notes Rechnerarchitektur 2. University of Mannheim, 1996-2002.

[85] Tom Shanley. PCI-X System Architecture. ISBN 0-201-72682-3, Addison Wesley, 2001.

[86] D. E. Culler, L. T. Liu, R. P. Martin and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *In Journal of IEEE Micro*, February, 1996.

[87] Mellanox. Introduction to Infiniband. White Paper, Mellanox, http://www.mellanox.com

[88] P. Druschel, L. Peterson and B. Davie. Experiences with a high speed network adaper: A software perspective. *In Proceedings of ACM SIGCOMM '94*, September 1994, pages 2-13.

[89] M. Fischer, U. Brüning, J. Kluge, L. Rzymianowicz, P. Schulz and M. Waack. Impact of Configurable Network Features in ATOLL. *In APSCC 2000, HPC Asia*, May 14-17, 2000, Beijing, P.R. China.

[90] http://www.hypertransport.org

[91] W. Gropp and E. Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Preprint MCS-P342-1193, Argonne National Laboratory, 1994.

[92] W. Gropp and E. Lusk. MPICH working note: Creating a new MPICH device using the channel interface. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1995.

[93] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *In IEEE Transactions on Computers C-21(9)*, pages 938-960, September 1972.

[94] Th. E. Anderson, D. E. Culler, D. A. Patterson. A Case for Networks of Workstations: NOW. em In IEEE Micro, Feb, 1995.

[95] Ch. Kurmann, M. Muller, F. Rauch and T. Stricker. Improving the Network Interfaces for Gigabit Ethernet in Clusters of PCs by Protocol Speculation. Technical Report No.339, Computer Science Department, ETH Zürich, 2000 (Extended Version of HPDC9 Paper).

[96] F. Seifert and W. Rehm. Reliably Locking System V Shared Memory for User Level Communication in Linux. *In proceedings of the IEEE International Conference on Cluster Computing CLUSTER2001*, Oct. 8-11, 2001, Newport Beach, California, USA.

[97] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. *In Hot Interconnects V*, August 1997.

[98] http://www.pccluster.org

[99] H. Tezuka, F. O'Carroll, A. Hori, Y. Ishikawa. Pin-down cache: A Virtual Memory Management Technique for Zero-Copy Communication. *In Proceedings of IPPS/SPDP 98*, March 98, Orlando, FL.

[100] T. Warschko, J. Blum and W. Tichy. On the Design and Semantics of User-Space Communication Subsystems. *In PDPTA 99*, Las Vegas, Nevada, 1999.

[101] M. Fischer. GMSOCKS - A Direct Socket Implementation on Myrinet. *In Proceedings of the IEEE Cluster 2001*, October 08-11, 2001, New Port Beach, CA, USA.

[102] M. Snir, S. W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. MPI - The Complete Reference: Volume 1, The MPI Core, 2nd edition. MIT Press, Cambridge, MA, 1998.

[103] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *In Parallel Computing*, 22(6):789–828, 1996.

[104] U. Brüning Vorlesung zur Rechnerarchitektur II. Chair of Computer Architecture, University of Mannheim, 1996-2002.

[105] T. von Eicken. Active Messages: an Efficient Communication Architecture for Multiprocessors. *In Ph.D. Thesis*, November 1993, University of California at Berkeley.

[106] M. Fischer and J. Simon. Embedding SCI into PVM. *In EuroPVM97*, Krakow, Poland, 1997.

[107] I. Zoraja, H. Hellwagner and V. Sunderam. SCIPVM: Parallel distributed computing on SCI workstation clusters. *In Concurrency: Practice and Experience*, Vol. 11, 1999

[108] J. Dongarra et al. PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing. MIT Press, Boston, 1994.

[109] IEEE Std for Scalable Coherent Interface (SCI). Inst. of Electrical and Electronical Eng., Inc., New York, NY 10017, IEEE std 1596-1992, 1993.

[110] H. Tezuka, A. Hori, Y. Ishikawa and M. Sato. PM: A Operating System Coordinated High Performance Communication Library. *In High-Performance Computing and Networking '97*, 1997.

[111] B. Bialek. http://www.db2mag.com/db_area/archives/2002/q1/bialek.shtml

[112] Direct Access Transport (DAT) Collaborative. http://www.datcollaborative.org

[113] A. Basu, T. von Eicken, M. Welsh. Low-Latency Communication over Fast Ethernet. *In Lecture Notes in Computer Science*, vol. 1123, 1996.

[114] A. Chien, M. Lauria, S. Pakin. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. *In Proceedings of the Technical Program of Supercomputing 1995*, December 1995.

[115] D.E. Culler, A.M. Mainwaring. Design Challenges of Virtual Networks: Fast, General-Purpose Communication. *In Proceedings of the 7 th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 119-130, Atlanta, GA USA, May 1999.

[116] E. Haydes. MM5 on SGI IA32 Clusters. *In Program for The Tenth Penn State/NCAR MM5 Users' Workshop*, June 21-23, 2000.

[117] http://www.windowsclusters.org/real_app_performance.htm

[118] Mukherjee and Hill. The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. *In Proceedings of HPCA98*, Feb. 1998.

[119] M. Fischer and J. Dongarra. PVM for Windows. *In Proceedings of CCC1997*, Atlanta, 1997.

[120] I. Foster. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. ISBN 0-201-57594-9, Addison-Wesley, 1995.

[121] T. Warschko. Efficient Communication in Parallel Computers. *PhD thesis*, University of Karlsruhe, 1998.

[122] IEEE 345, 47th Street New York. IEEE Standard for Scalable Coherent Interface (SCI), 1993.

[123] E. Rehling. SThreads: Multithreading for SCI clusters. *In Proceedings of SCI Europe 1999*, Toulouse, France, 1999.

[124] W. Gropp, E. Lusk and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi/mpich/papers/mpicharticle/paper.html

[125] K. Magoutis et al. Structure and Performance of the Direct Access File System. *In Proceedings of the 2002 Usenix Annual Technical Conference*, Monterey, CA, USA, 2002.

[126] T. X. Jakob. Multilevel Optimization of Parallel Applications Utilizing a System Area Network. *Master Thesis*, University of Mannheim, Chair of Computer Architecture, 2002.

[127] TecChannel. So funktioniert TCP/IP. http://www.tecchannel.de/internet/209/9.html

[128] A. Jones and A. Deshpande. Windows Sockets 2.0: Write Scalable Winsock Apps Using Completion Ports. MSDN Magazin, October 2000.

[129] J. L. Hennessy and D. A. Patterson. Computer Organization and Design. 2nd Edition, San Francisco: Morgan Kaufmann Publishers, 1997.

[130] D. Anderson, J. S. Chase, S. Gadde, A. Gallatin, and K. Yocum. Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet. In Proc. Usenix Technical Conference, New Orleans, LA, June 1998.

[131] V. Pai, P. Druschel and W. Zwaenepoel. IO-Lite: A Unified I/O Buring and Caching Scheme. In Proc. of Third USENIX Symposium on Operating System Design and Implementation, New Orleans, LA, February 1999

[132] A. Silberschatz and P. B. Galvin. Operating System Concepts. Addison-Wesley, ISBN 0-201-54262-5, 1998.

[133] IBM Eserver. http://www.ibm.com/servers/eserver/pseries/hardware/entry/sales_shee June 25, 2002.

[134] A. Petitet, R. C. Whaley, J. Dongarra and A. Cleary. http://www.netlib.org/benchmark/hpl

[135] Stevens W. Richard. TCP/IP Ilustrated I. The protocols. Addison-Wesley, 1994.