Karlsruher Institut für Technologie

Porting LibRIPC to iWARP

Bachelorarbeit
von

cand. B.Sc. Felix Fereydun Pepinghege

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuender Mitarbeiter:	Dr. Jan Stoess

Bearbeitungszeit: 25. Mai 2012 – 24. September 2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die Regeln des Karlsruher Instituts für Technologie (KIT) zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, den 24. September 2012

Abstract

Cloud computing has become a major economical factor in the recent development of computer systems. Companies tend to draw computational power for their data processing from the cloud, instead of hosting their own servers, in order to save costs. The providers of cloud systems run huge and cost efficient data centers, profiting from economies of scale. In order to further reduce the costs, these data centers currently move to more power efficient systems.

Many applications that are now used in the cloud were not created with a cloud environment in mind, but have been "moved" to the cloud. These applications usually use TCP/IP for their intercommunication mechanisms, which is the de-facto standard for current applications in the Internet. Unfortunately, these TCP/IP based implementations rely on the Berkeley socket API, which does not match the demands of power efficient systems. Sockets introduce much CPU involvement, taking away precious computational time from the real applications.

Several specialized network architectures, such as InfiniBand, overcome this issue. They make use of offloading techniques, such as RDMA, in which the network hardware takes over the responsibility of protocol processing and I/O. As a drawback, the equipment for these fabrics is usually expensive, due to their high specialization. Furthermore, the corresponding interfaces for these architectures are extensive, making it expensive to adapt existing applications.

LibRIPC is a network library introduced to overcome the efforts that are accompanied by the usage of these architecture's interfaces. It provides a neat, message based interface, which abstracts from any specifics of the underlying network architecture. This provides ease of integration and portability to other network fabrics, and yet does not sacrifice performance. Currently, there exist a prototype implementation of LibRIPC for InfiniBand.

In this thesis, we present our port of LibRIPC to iWARP, which enables the library for its use over Ethernet. Ethernet is one of the most cost efficient network fabrics and provides capabilities for high performance networking since the upcoming of standards that specify data rates of 10 Gbit/s and above. iWARP is a protocol stack that is based upon TCP/IP, thus compatible for use in Internet based WANs. Yet it is completely independent from the socket API and its correspond-

ing drawbacks.

We evaluated our implementation in terms of latency and data throughput, and achieved promising results. Despite using software based iWARP instead of better performing iWARP enabled hardware, LibRIPC over iWARP outperforms TCP sockets by far. We believe that our approach can achieve even better results on appropriate hardware.

Deutsche Zusammenfassung

Im Rahmen der aktuellen Entwicklung von Computer Systemen ist "Cloud Computing" ein signifikanter wirtschaftlicher Faktor geworden. Mit der Absicht Kosten einzusparen, neigen Firmen dazu, ihre IT Ressourcen aus der Cloud zu beziehen, anstatt eigene Server zu betreiben. Die Anbieter von Clouds betreiben riesige Rechenzentren die von Skaleneffekten profitierend, sehr kosteneffizient arbeiten. Um die Ersparnisse weiter zu vergrößern, setzen diese Rechenzentren zunehmend auch auf Energie-effiziente Systeme.

Viele Anwendungen die mittlerweile in der Cloud betrieben werden sind ursprünglich nicht für diese Systeme konzipiert worden, sondern wurden erst nachträglich in die Cloud integriert. In der Regel verwenden diese Anwendungen TCP/IP für ihre Kommunikation, was dem Quasi-Standard für aktuelle Internet-Programme entspricht. Diese Implementierungen auf Basis von TCP/IP verwenden Berkeley Sockets als Programmierschnittstelle, die für Energie-effiziente Systeme aber ungeeignet sind. Berkeley Sockets benötigen viel Prozessorzeit, die dadurch nicht mehr für die eigentlichen Programme zur Verfügung steht.

Es gibt viele spezialisierte Netzwerkarchitekturen, zum Beispiel InfiniBand, die dieses Problem lösen. Diese Technologien nutzen Mechanismen wie RDMA, in denen die Netzwerkgeräte die Verantwortung für Protokollverarbeitung und Datenübertragung selbst übernehmen. Ein Nachteil sind die hohen Kosten für Geräte, die durch die hohe Spezialisierung der eingesetzten Hardware entsteht. Des weiteren sind die Programmierschnittstellen dieser Architekturen häufig ausufernd, und erschweren eine Integration in bereits existierende Anwendungen.

Die Netzwerkbibliothek LibRIPC, versucht den Nachteil dieser unhandlichen Programmierschnittstellen auszugleichen. Sie stellt eine übersichtliche, Nachrichten-orientierte Schnittstelle zur Verfügung, die zudem von allen Architektur-spezifischen Eigenschaften abstrahiert. Auf diese Weise ist LibRIPCeinfach zu integrieren und kann auf mehrere unterschiedliche Architekturen portiert werden, ohne dabei auf Leistungsfähigkeit verzichten zu müssen. Zum jetzigen Zeitpunkt existiert eine prototypische Implementierung auf Basis von InfiniBand.

In dieser Arbeit stellen wir unsere Portierung von LibRIPC auf iWARP vor, die unsere Bibliothek auf Ethernet lauffähig macht. Ethernet ist eine der kosten-

effizientesten Architekturen, die seit der Spezifizierung von Standards für Datenraten von 10 Gbit/s und mehr, auch eine hohe Leistungsfähigkeit erreicht. iWARP ist ein Protokollstapel der auf TCP/IP aufsetzt, und damit kompatibel zu Internet-basierten Weitverkehrsnetzen (WANs) ist. Nichts desto trotz ist es komplett unabhängig von Sockets und den einhergehenden Nachteilen.

Wir haben unsere Implementierung hinsichtlich der Datenrate und der Verzögerungszeiten untersucht, und dabei viel versprechende Resultate erhalten. Obwohl wir nur auf Software basiertes iWARP, anstatt auf performantere iWARP-kompatible Netzwerkkarten zurückgegriffen haben, hat unsere Implementierung die Leistungsfähigkeit von TCP Sockets deutlich überstiegen. Wir glauben dass unsere Arbeit sogar noch bessere Resultat liefert, wenn die entsprechende Hardware eingesetzt wird.

Contents

Chapter 1

Introduction

In the recent evolution of computer systems the concept of *cloud computing* [34, 36] has become more and more prevalent. Especially in the economy, the cloud has been detected as a possibility to run the company's data processing more cost efficient [13, 32]. Hosting the needed IT services on its own is a costly, because the servers and the network equipment must be able to bear the peak demands, which often exceed the average demand by far [33]. This leads to time spans of underutilization; the company pays for resources that it does not need at the time. Cloud computing services, by contrast, often offer "pay-as-you-go" charging, which allow the costumer to allocate and pay for more computing resources only when needed. Providers of cloud systems are capable of running their systems much more cost efficient than individual companies. They benefit greatly from economies of scale in terms of nearly all resources, for instance network technology, storage technology and manpower for administration [5].

However, in order to achieve cost efficiency, cloud systems need to be highly optimized. One aspect that contributes largely to the overall costs is the power consumption, leading to a higher relevance of energy efficiency in cloud systems [37, 71]. As a consequence cost- and power-efficient processors, such as ARM or Intel Atom chips [10], are starting to substitute the commodity computers, which are currently used frequently [37]. These CPU cores provide less individual computational power and are therefore applied in larger numbers over which the computational load is distributed.

The communication between these nodes is a crucial resource. Cloud applications put various demands on the interconnection fabric, as they require a lot of network communication of different natures. Client-server applications, such as web or file servers, often need high bandwidth in order to exchange large data rapidly. On the other hand, applications such as distributed databases, primarily exchange small messages for synchronization purposes, which tend to demand low latency. All in all, we observe a demand for high performing network transfer

which does not put too much pressure on the scarce CPU resources.

Arguably, the most common network protocols are the members of the Internet protocol family [14], which date from the 1970's and are still the standard protocols in today's Internet. These protocols, that is TCP/IP or UDP/IP, mostly over Ethernet, are usually implemented via Berkeley Sockets [42] using the respective socket API. Unfortunately, the socket API in general, and TCP in particular, are not suitable for the herein before mentioned demands that we observed [41]. Network communication with TCP/IP heavily involves the CPU because of its implementation as system calls [58], copy overhead [15], and protocol processing [26]. Additionally the copy overhead burdens the system's memory bus [7].

These drawbacks of TCP/IP are long known, and several network architectures were developed to overcome these issues, such as Myrinet [8] and InfiniBand [59]. However, the usage of these architectures is not trivial, as the corresponding interfaces, for example the InfiniBand Verbs [39], are often extensive and hard to use. Re-writing an application from Berkley sockets to such interfaces is therefore expensive.

In order to combine the benefits of high performance network architectures and the simplicity of the socket API, [45] introduced LibRIPC, a library for remote inter process communication. The library provides a neat, message based interface, which abstracts from all specifics of the network architecture. Therefore, LibRIPC provides ease of integration without sacrificing performance, and portability to other network architectures. The interface distinguishes two messages types, one focusing on high data throughput and one with focus on low latency. Currently, there exists a prototype implementation of LibRIPC, which uses InfiniBand and yielded promising results.

Despite the impressive performance results, which are achievable with InfiniBand [77], using Ethernet may be desirable. Highly specialized architectures, such as InfiniBand, are relatively expensive for several reasons. On the one hand, the specialized hardware increases the costs for the equipment, for a rather simple reason. The demand for such equipment is rather low, the output figures of the production accordingly small. This makes it hard for vendors to lower the costs of production. The high acquisition costs especially preponderate because every part of the hardware is affected: network adapters, switches and cables. On the other hand, dealing with specialized hardware requires specialized know how, which in turn leads to higher expenses in terms of manpower. As opposed to that, Ethernet is an ubiquitous fabric, which does not suffer from all these drawbacks [26,53,55]. In fact, Ethernet is so inexpensive, that most modern desktop computer systems come with 1 Gigabit Ethernet links by default. Additionally, Ethernet equipment that allows for relatively high performance is downward compatible to slower and therefore cheaper variations. It is therefore possible to build a homogeneous network, which varies in its levels of performance, according to the corresponding

needs [46].

With the upcoming of 10 Gigabit Ethernet and beyond, Ethernet has become a target architecture to provide high performance networking, with several approaches token (see Section 2.1). One of these approaches is the *Internet Wide Area RDMA Protocol* (iWARP) [2, 55, 63], a protocol stack (discussed in detail in Section 3.1.1) on top of TCP/IP. iWARP provides high performance networking by implementing *Remote Direct Memory Access* (RDMA) [65], that is, direct data transfer from the user memory of one machine into the user memory of the other machine, bypassing the kernel. Additionally, hardware implementations are available [19, 43, 57], which offload the complete iWARP protocol stack to the hardware and achieve remarkable performance [17, 21, 25, 62, 69]. Besides the promising performance results of network adapters with iWARP capability, the fact that iWARP bases on TCP/IP is a significant advantage, especially in the context of cloud computing, where the Internet is right at hand. In order to use iWARP, intermediate hardware, such as switches and routers, need neither be modified for nor be aware of iWARP. Endpoints of iWARP based communication can be deployed in any existing TCP/IP based network, such as the Internet, and work out of the box [2].

However, from the programmer's point of view, iWARP brings the same drawbacks as InfiniBand, that is, an extensive, hard to handle interface [35]. In this thesis, we present our porting of LibRIPC to iWARP, which provides Ethernet compatibility for LibRIPC. We believe that this way, we are able to combine the benefits of the neat and easy to integrate interface of LibRIPC, with the advantages of iWARP, in terms of cost efficiency and flexibility, without loosing the high performance that InfiniBand provides. With this extension of the pool of network architectures supported by LibRIPC's, we can make the usage of our library much more lucrative, and proof its portability, which was a major design goal, as stated in [45].

In order to evaluate our approach we implemented iWARP support in LibRIPC, and conducted two experiments, which measure the performance of our implementation. Because we lack iWARP-enabled network adapters, we applied the experiments on software based iWARP over InfiniBand network adapters. Despite this throttling setup, we found that our implementation is able to keep up with InfiniBand, both in terms of latency and throughput. Additionally, we were able to demonstrate the portability of LibRIPC, by using the same applications for both iWARP and InfiniBand, without any modifications needed.

The rest of the thesis is structured as follows: We begin with an overview of the background and related work in Section 2, before we turn to the design of our approach in Section 3. The evaluation of our design is discussed in Section 4, before we conclude the thesis in Section 5.

Chapter 2

Background and Related Work

Over the last two decades, the Internet became a huge part in every day's life and established itself as an important factor for the economy and for people's private life. In this time, the Transmission Control Protocol (TCP), the User Datagram Protocol (UDP), and the Internet Protocol (IP) became the de-facto standards for communication over the Internet, and are mostly deployed on top of Ethernet. Despite huge changes and development in the network architecture, these protocols have not changed much over the years. However, these protocols are usually used via the Berkeley socket API [42], which suffers from several drawbacks, in the context of high network traffic.

Several interconnects exist that do not have these drawbacks, such as InfiniBand [39] or the Blue Gene interconnection fabric [12, 30], but they are generally much more expensive than Ethernet, provide only an extensive interface or are specialized for the context of High Performance Computing (HPC). With the emergence of 10 Gigabit Ethernet and above, the possibility of implementing high performance networking over the cost efficient Ethernet fabrics arises [26, 53, 55]. This is a great opportunity for cloud computing, as most of the applications that currently run in the cloud are not implemented for those high performing network architectures, but are implemented on top of TCP/IP, that commonly runs on top of Ethernet. Unfortunately, TCP/IP is not suitable for high performance, as it implies too much overhead.

In order to analyze the overhead that is introduced by the use of TCP over sockets, we take a look at the path that data has to take in order to get transferred. The socket API is a collection of system calls; protocol processing itself is done in the kernel. As a first step, the kernel copies the data from its position in user space into a buffer that resides in kernel space. This copying process burdens the processor, as it must be done by the CPU [15]. The kernel buffer in which the data is saved is then extended with protocol information, such as sequence numbers and checksums, which the CPU must generate. For the purpose of correct

TCP protocol processing, the kernel must maintain a series of mechanisms, such as timers and buffer handling. In general, the implementation uses general purpose data structures for these tasks, in order to keep the number of in-kernel data structures reasonable. However, these data structures are not optimized for TCP's needs and thus prevent an efficient implementation [15]. When a TCP packet is complete, it is copied to the network adapter, usually via a DMA transfer.

On the receiving end, we observe the same overhead in the opposite direction. However, if data arrives continually at one host, the network adapter continually issues interrupts, which in turn lead to context switches that introduce direct overhead, such as dispatching, and indirect overhead, such as cache poisoning. In total, data transmission via TCP requires at least four copy processes that are in contention for memory bus usage [29].

Additionally, TCP is a byte-stream oriented protocol, that is, message borders are not preserved and must be recovered by the receiver. In case of large data, such as files, the payloads arrives in multiple pieces, whose exact number the receiver does not generally know. Therefor, the receiver has to constantly issue calls to the receive function.

As the development of processing power and of the memory bus could not keep up with the network technologies, this overhead becomes a bottleneck. Altogether we observe four factors which combine to socket or TCP related overhead, which needs to be avoided:

i) Data copying
ii) Kernel invocations
iii) Protocol processing
iv) Byte-stream orientation

One technique to overcome these issues is iWARP, which uses a protocol stack above TCP/IP, to bypass the kernel and to avoid multiple copying of data. The fundamental concept that is implemented by this protocol stack is RDMA, that allows to transfer data directly from its position in the user-level memory onto the network adapter, which performs the protocol processing and transfers the data directly into the user-level memory of the receiver. As this is done by using TCP/IP, this transfer is not limit to the local network, but can be used over the Internet [2], and therefore introduces a clear benefit in the context of cloud computing. This possibility does not sacrifice the performance within a local network; iWARP's performance results are impressive and can bear comparison with highly specialized network architectures, such as InfiniBand [17, 21, 25, 62, 69].

However, iWARP is not the only approach to bring high performance networking to Ethernet. We present some alternatives in the following, and evaluate in how far they are suitable as a target for a porting of LibRIPC. Afterwards, we discuss the possibility of software based implementations of iWARP, its benefits, and its drawbacks.

2.1 High Performance Solutions for Ethernet

In this thesis, we describe the porting of LibRIPC to iWARP with the objective to fully use the performance capabilities of modern Gigabit Ethernet. However, iWARP is not the only approach to bring high performance networking to Ethernet fabrics. This section provides a selection of these approaches.

2.1.1 NetSlice

NetSlice [47, 48] is a kernel-side abstraction of network transfer via raw layer 2 sockets, implemented for Linux. It is not restricted to a specific underlying network architecture – NetSlice works out of the box with every network device driver for Linux. The idea of NetSlice is to provide a tight coupling between hardware resources and user-level application, to minimize in-kernel overhead. Furthermore, applications are able to bind queues of the network cards to designated CPU cores. This prevents resource contention of multiple kernel threads which want to access the same resource from different cores. With this mechanism the application can achieve a linear scalability to the number of cores and thereby bail out the bandwidth provided by the network adapter. Furthermore, NetSlice provides the possibility to reduce the number of kernel invocations by providing a batch mechanism, which allows to add multiple send/receive operations to the processing queue via one system call.

NetSlice is able to achieve good results in terms of network throughput, and scales very good for multiple cores and network adapters. However, it operates an raw layer 2 frames, what makes it very uncomfortable to handle. Additionally, it is the responsibility of the programmer, to make sure that the application uses a well performing binding between a network adapter and a CPU core. NetSlice is more suitable for packet processors, such as user-level network stacks, packet filters or software-based routers, which can benefit from the provision of layer 2 frames.

2.1.2 RDMA over Converged Enhanced Ethernet

RDMA over Converged Enhanced Ethernet (RoCEE) [16] is essentially InfiniBand over "lossless Ethernet". CEE describes an Ethernet-based network in which all used switches support several features, which are marked optional for standard Ethernet. These features include Enhanced Transmission Selection and the Data Center Bridging Exchange Protocol as described in the 802.1Qaz specification, Priority-based Flow Control as of 802.1Qbb and Congestion Notification as standardized in 802.1Qau [56]. The set of these specifications enables lossless networking, support of classes of service and congestion management for Ethernet.

InfiniBand requires exactly these features for an efficient operation [16].

As RoCEE uses InfiniBand protocols on top of Ethernet, the InfiniBand Verbs are the native interface to the application. Thus, we expect RoCEE to be easily integrated in LibRIPC, making it a very interesting option for porting. Additionally, RoCEE is the third target system of OFED (see 3.3) next to InfiniBand and iWARP. An implementation of LibRIPC in the future may be beneficial, as RoCEE may well experience increasing relevance in the context of HPC.

2.1.3 Ethernet Message Passing

Ethernet Message Passing (EMP) [68] is a messaging protocol introduced in 2001, which provides zero-copy network transfer with kernel bypassing. Target systems of EMP are programmable network cards which allow it to replace the complete firmware of the card. All protocol processing is realized in a newly written firmware and requires no CPU involvement as DMA is used to transfer data between main memory and the network card. Despite providing low latency, the very small protocol puts pressure on the programmer, who is responsible for all packet handling but reliability, which is guaranteed by EMP. We find EMP not to be very suitable for our purposes, as it requires fully programmable network cards and provides few packet handling. Additionally, EMP is everything but common and has played no significant role in networking.

2.1.4 Open-Mx

Open-Mx [31] is an emulator of the Myrinet Express application interface. It is a kernel module, operating on top of Linux's Ethernet layer, thus allowing to execute applications written for Myrinet hardware on Ethernet networks. The major drawback of this design is that Open-Mx does not allow kernel bypassing. However, zero-copy transfer is supported by a trade-off mechanism, which triggers the zero-copy transfer if the passed message is large enough to introduce more overhead by copying than by pinning of the memory page. With these mechanisms, Open-MX is able to effectively exhaust 1 Gigabit Ethernet cards, but does not reach more than 6 Gigabit/s when 10 Gigabit Ethernet is employed. As Open-Mx primarily addresses interoperability, sophisticated performance for Ethernet is not a main goal. It is therefore no reasonable option as a porting-target of LibRIPC.

2.2 Software based iWARP

iWARP is solely a protocol stack and puts no demands on the underlying hardware. The only requirement is, that both endpoints of the communication are

aware of iWARP, and support the corresponding protocols. Basically, that means that the entire protocol stack can be implemented in software and run on top of any TCP socket implementation. Such an implementation on top of the regular socket API does not provide any advantages over a direct use of the sockets. However, software based iWARP can provide benefits, and implementations do exist [6, 22, 79].

One benefit of software based iWARP is that machines which run iWARP compliant hardware, can profit from this hardware while communicating with machines that run on software driven iWARP. For instance, a server may provide a service that is not very performance critical in terms of network throughput or latency, but requests of clients require computation, such as dynamic web sites. The CPU offloading that is provided by iWARP, can make the server scale much better, as no computational power is wasted on the network transfer. This does not lead to any advantages on the client's side, but the server can handle more requests at a time.

Additionally, benefits can be achieved when the protocol stack is implemented in the operating system's kernel. This way, the application can reduce the number of system calls, what in turn can lead to significant performance improvement [58].

Such an approach is taken in [79]. Here, the architecture is divided in a kernel module that implements the protocol stack, and user-level library that provides the interface for the application [50–52,54]. Because of the in-kernel implementation, this approach achieves better performance than TCP sockets, even on ordinary Ethernet adapters, but can not provide the same degree of CPU offloading, as hardware based iWARP.

In [22], two approaches are provided; one that implements the protocol stack entirely in user level, and one that implements it entirely in the kernel. While the user-level protocol stack is accessed via a library [23], the kernel-level implementation provides interaction via system calls [24]. Both approaches do not provide significant savings in terms of the number of necessary system calls. Therefore neither of the two approaches yields significant performance improvements, in comparison with TCP sockets.

A third approach is introduced in [6] and provides a total of four different variations with different degrees of user-level offloading. "User-level iWARP" runs the protocol stack on top of ordinary sockets, whereas the "Kernel-level iWARP" is implemented in the kernel on top of TCP/IP stack. "Software iWARP" implements the protocol stack on top of offloaded TCP and the fourth variation uses a network adapter that is capable of iWARP. Applications interact with the iWARP subsystem via an extended socket interface, which overloads the standard *libc* library. With this interface, sockets can be set set to "iWARP mode". If a socket is set to this mode, the library translates the calls to the respective subsys-

tem that is currently in use. If a socket is not set to this mode, the call is simply passed to the traditional socket interface.

Chapter 3

Design

In this chapter we present our approach to integrate iWARP in LibRIPC. For this purpose we first describe the structure of iWARP, to explain how the protocol stack operates and how it achieves its good performance. Next, we present the major design goals of LibRIPC and which mechanisms are used to achieve them. We than discuss which techniques available through iWARP can serve as appropriate mechanisms for our purpose. Our decisions are then named and reasoned.

3.1 iWARP

iWARP is a protocol stack invented to bring high performance networking to Ethernet fabrics. The prerequisites that we discussed in Chapter 2 are addressed with complementary concepts.

Protocol offloading is implemented via a *TCP Offload Engine* (TOE) [26, 53, 78]. A TOE performs all processing of the TCP/IP stack in hardware, that is, on the network card. This way, the CPU is not loaded with calculations of sequence numbers, (de-)allocation of buffers or establishing of timers; things that account for a large part of TCP's overhead [15]. Network cards that are compliant to iWARP, are called *RDMA enabled Network Interface Controller* (RNIC). However, TCP offloading provides noticeable performance increase, but it does shed only a part of the overhead [29, 55].

In order to deal with the other prerequisites – zero copy and kernel bypass – iWARP implements the *Virtual Interface Architecture* (VIA) [18], a concept for standardised efficient cluster communication. VIA introduces the fundamental concept of RDMA, which implies zero copy and kernel bypass. These concepts are implemented by the establishment of virtual interfaces, which abstract from the Network Interface Controller (NIC), that is the network card [11]. A *Virtual Interface User Agent* (VI User Agent) embodies the abstraction and provides an

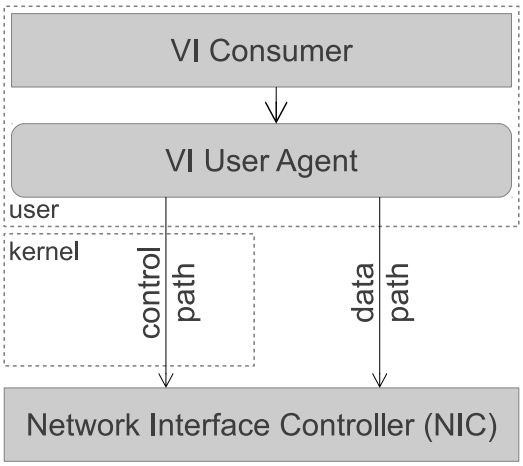


Figure 3.1: A simplified illustration of the Virtual Interface Architecture (VIA). The VI Consumer uses the VI User Agent, which accesses the NIC either via a control path trough the kernel (for management purposes), or via a data path (for transmission purposes), bypassing the kernel.

interface for virtual-interface aware applications, the so called *VI Consumers*. In order to manage the NIC, the VI User Agent uses a control path to the NIC that passes the kernel. For data transfer, the VI User Agent bypasses the kernel through a data path. A simplified overview is depicted in Figure 3.1. The VI Consumers cherishes the illusion of having a dedicated network interface, of which it has full control. This illusion provides the VI Consumer with the possibility of directly accessing the NIC, thus "virtualizing" the network interface. Originally designed to overcome proprietary interfaces for System Area Networks[1], VIA's concepts were adopted by several fabrics, such as InfiniBand and iWARP, and are now applied in several use cases, such as network storage and remote inter-process communication. [11].

iWARP implements VIA based upon the Internet protocol stack. That is, on layer 3 of the ISO OSI reference model [40], iWARP uses IP. At OSI layer 4, one of two specified protocols can be used: The Stream Control Transmission Protocol (SCTP) or TCP [29,35,51]. Although SCTP performs better in a message passing context [44] and is more suitable to iWARP's needs as TCP [75], all available hardware implementations of iWARP, that we are aware of [19, 43, 57], use TCP. A detailed view on iWARP's protocol stack and on SCTP is provided in Section 3.1.1.

[1]System Area Network is a collective term for high performance communication fabrics for clusters.

The idea of RDMA is to avoid or at least minimize CPU involvement in the transmission process. In this concept the hardware is performing the major part of the work itself, the user application only has to launch the transmission process (just like "regular" DMA and hence its name). As soon as the RDMA controller knows the data source and the data sink on the remote end, all data is directly transferred from the user-space memory of the sender into the user-space memory of the receiver. This helps increasing the network performance in two ways. On the one hand, the data must not be copied from user space into kernel space before the transfer can start. This obviously saves the time needed to perform the copy process. Furthermore, it reduces system-bus allocation and CPU involvement, as this transfer is always performed by the processor [15]. On the other hand, RDMA avoids occupation of applications. When large pieces of data are transferred via sockets, the receiver has to issue multiple read operations, because TCP fragments the data and sends it in multiple pieces instead of one message. Not only does this keep the application active, it additionally invokes the kernel, hence causing protection-domain crossings.

A basic distinction between a RDMA transfer as opposed to a socket transfer is how both communication partners are involved in the process. When using the socket interface, both sides of the communication have an active role, that is, the sender performs a send system-call whereas the receiver performs a (or multiple) receive system-call(s). This way the receiver is immediately informed when data arrives. In case of a RDMA transfer, only one side is active – either the sender or the receiver. If the sender occupies the active role, it performs a RDMA write, that is, it directly writes the data in the receiver's memory. Otherwise, the receiver performs a RDMA read, that is, it directly reads from the sender's memory. In both cases the passive side of the communication has no indication of this transfer. In general, it is neither informed when its communication partner starts, nor when it completes the transfer. Instead, the applications must communicate the transmission explicitly.

However, to allow for RDMA to the remote communication partner's memory, the passive side is not utterly uninvolved. The data has to reside in a so called *memory region*, where it can be directly accessed by the network hardware. This usually requires a registration of the memory with the hardware. In this registration process, the hardware learns the address and assigns a key to it. Such an address/key pair is called *Steering Tag* (STag). The key serves as a precaution to prevent untrusted remotes from accessing memory by guessing the address. Furthermore, the memory belonging to the MR has to be pinned, that is, it has to be prevented from being swapped out by the operating system. This is necessary to ensure the consistency of the memory on which a remote communication partner performs read or write operations. Before a RDMA transfer can start, both the source and the target memory must be registered as a memory region. Addition-

ally, the active part of the communication must know the STag of the remote's memory region. It is the responsibility of the user to provide a way to exchange the necessary information to the communication partner – a process called *buffer advertisement*.

3.1.1 The iWARP Protocol Stack

As already mentioned, iWARP is not one protocol, but a protocol stack, which distributes different jobs to individual protocols. Above Ethernet and TCP/IP, three further protocols are used, from which one becomes unnecessary, if SCTP is used. The protocol stack is depicted in Figure 3.2, in comparison to TCP/IP (without TOE) with the socket API. One can see the protocol offloading and kernel bypassing of iWARP.

The **Remote Direct Memory Access Protocol** (RDMAP) [35] is the topmost protocol in this stack. RDMAP is specified as a semantic interface in the form of a verbs specification. Verbs do not specify the syntactical interface (i.e., names of methods, order of parameters, and return value) but they precisely describe the actions that a RDMAP implementation must support, including their behaviour, the set of accepted parameters and error handling. Verbs may be declared optional, in which case an implementation does not have to support it. However, if an optional verb is implemented, it must match the specified behaviour accurately.

The verbs specifications of RDMAP and InfiniBand share a very large subset. Almost all fundamental concepts are identical, such as protection domains, queue pairs, and memory regions. Differences only appear if RDMAP and InfiniBand differ in their provided functionality. For instance, RDMAP does not support multicasting, whereas InfiniBand does. Any specifications of InfiniBand regarding this are obviously not included in the RDMAP verbs.

In RDMAP, a connection to a remote communication partner is managed via *queue pair*s, one for every endpoint of the connection. A queue pair consists of a *send queue* and a *receive queue*, and corresponding *completion queue*s. The verbs consumer interacts with a queue pair by posting *work requests* on the send/receive queues and by receiving *work completions* on the completion queues.

In order to send a message, the consumer creates one or more work requests, which is filled with all necessary information. These information contain the message type, flags, address and size of the source buffer and – in case of a RDMA operation – the STag of the remote memory region. When the transfer is completed, RDMAP generates a work completion on the send queue, if the user specified this behaviour. On the remote side, a work completion is generated on the receive queue, if the sender's work request was configured accordingly and the receiver

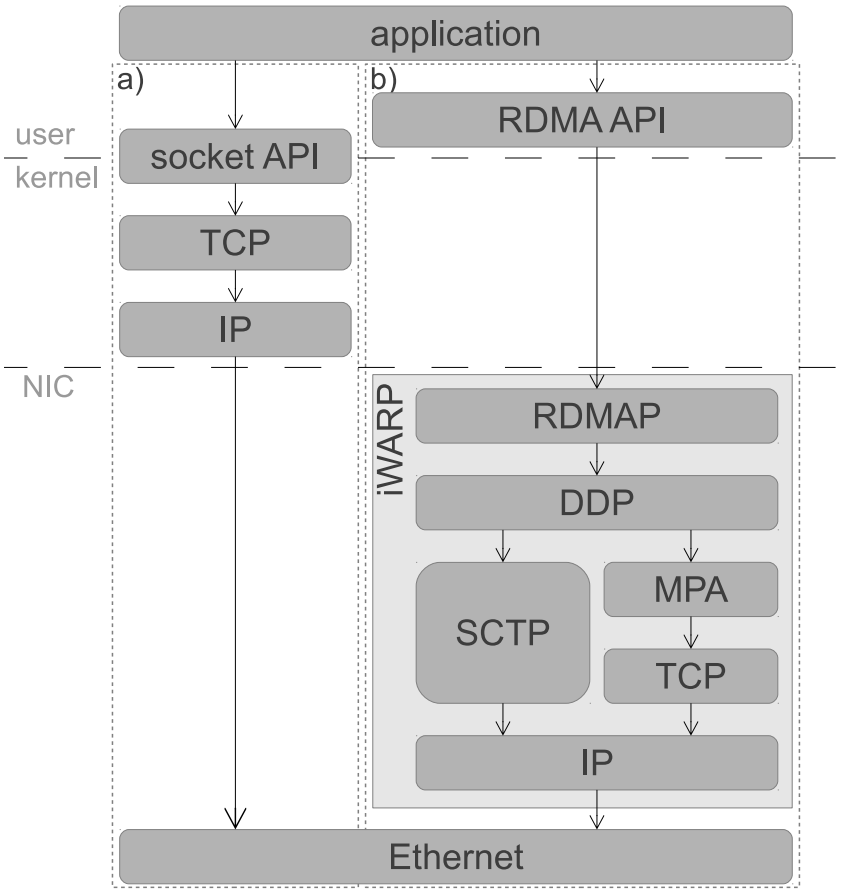


Figure 3.2: The structure of b) the iWARP protocol stack compared to a) the socket API. The RDMAP API completely resides in user space and uses the iWARP stack, which in turn is implemented in hardware (i.e., on the NIC). The socket API resides on the border between user and kernel space, as the API consists of system calls. All protocol processing of TCP/IP is done in kernel space.

posted a receive work request on its receive queue. All send and receive operations are inherently performed asynchronous; a call to post a send or a receive work-request returns immediately. However, it is up to the consumer to decide whether he wants to wait blocking for a work completion, poll the completion queue occasionally, or receive no work completion at all.

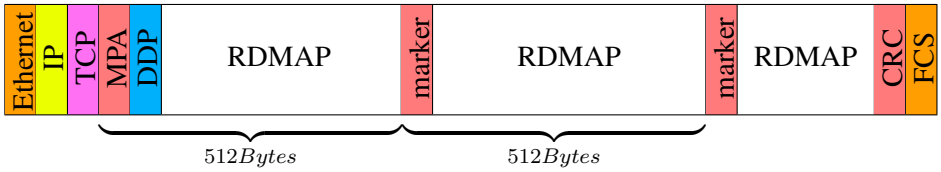
The **Direct Data Placement Protocol** (DDP) [66] is the lower layer protocol (LLP) of RDMAP. That is, it resides directly underneath of RDMAP in the protocol stack. DDP enables the zero-copy data transfer, which is essential for RDMA.

With TCP, it is always necessary to buffer incoming messages before they can be made available for the user application, because it is not guaranteed for the packets to arrive in order. Changing routes or lost and then resend packets can lead to such out-of-order arrival. DDP divides the data into chunks which fit in a single segment of the respective LLP. These chunks are enriched with information about the target memory region and the offset of the contained data within the memory region. With this information, the payload of the network packet can be placed directly in the correct place, despite possible out-of-order arrival.

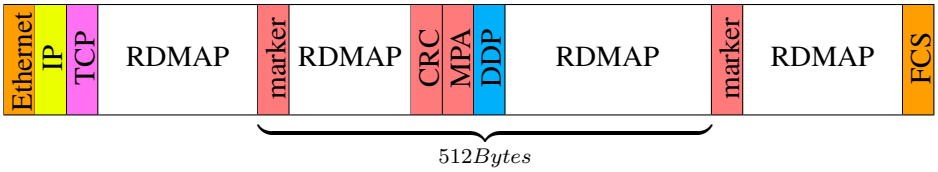
Marker PDU Aligned Framing for TCP (MPA) [20] is a framing protocol of the iWARP stack. That is, MPA aligns DDP chunks to TCP packets. PDU stands for protocol data unit and denotes the data that a protocol receives from its upper layer protocol. In the context of iWARP, the PDU of MPA is the DDP chunk. MPA resides between DDP and TCP and allows the receiver to rediscover message boundaries inside TCP's byte stream. Ideally, MPA and TCP coordinate to make every TCP packet start with one MPA frame [20], as depicted in Figure 3.3(a). With this alignment, every network packet can be processed on the fly, as the position of the DDP header is known and the target memory region of the data can be read immediately. If, for some reason [54], this alignment between MPA and TCP gets lost, the position of the MPA header has to be recovered. For this purpose, MPA adds markers in a periodic fashion to the byte stream. Figure 3.3(b) shows an example of an Ethernet frame in which the headers are not aligned. Every marker stores a pointer to the last MPA header. As the markers are distributed periodically, their position can always be recalculated using TCP's sequence number as an orientation. The example in Figure 3.3(b) shows that in a case of misalignment, DDP chunks can be spread over multiple TCP packets. This leads to the necessity of buffering a packet, if a corresponding DDP header resides in a preceding packet, which has not yet arrived. In order to accomplish zero-copy data transfer, it is therefore crucial that MPA provides header alignment.

A second responsibility of MPA is to implement a reliable error detection. This is necessary to guarantee correct data transfer. iWARP can neither rely on the frame check sequence (FCS) of Ethernet, nor on the TCP checksum. The Ethernet's FCS does not cover errors that may occur during TCP/IP processing in routers or gateways. These errors must then be detected by the TCP checksum, which in turn has betrayed weaknesses that can lead to corrupted data [74]. MPA adds a reliable checksum to the end of its frame, as depicted in Figure 3.3, to compensate this issue.

The **Stream Control Transmission Protocol** (SCTP) [73] is a layer 4 protocol, which serves as an alternative to TCP. Transmission over SCTP is message rather



(a) iWARP over TCP/MPA with aligned headers: The MPA header directly follows the TCP header. Both markers point to this MPA header.



(b) iWARP over TCP/MPA with unaligned headers: The MPA header is placed at an arbitrary position inside the TCP packet. The first marker points to the MPA header in the preceding packet, the second to the MPA header in this packet.

(c) iWARP over SCTP: The network packet gets along with five headers and the RDMAP is not fragmented.

Figure 3.3: The structure of an iWARP network packet over (a) aligned TCP/MPA, (b) unaligned TCP/MPA, and (c) SCTP. Colored fields represent protocol data. FCS stands for frame check sequence (the error detection mechanism of Ethernet).

than byte-stream oriented. That allows the upper layer protocol (ULP) – DDP, in the case of iWARP – to send its packets atomically. For instance, if DDP divides a message into two chunks of 512 Bytes each, and passes them to SCTP, it is guaranteed that SCTP transfers these chunks in two individual packets [70]. TCP on the other hand has no guaranteed behaviour in such a case. The chunks may be send together in one packet or split up at any position. Additionally, SCTP provides strong and reliable checksumming. As these properties match the functionality of MPA, SCTP can substitute TCP and MPA, reducing the size of the iWARP stack, as depicted in Figure 3.3(c). Theoretically, that makes SCTP the protocol of choice for iWARP. However, despite being implemented for a variety of operating systems (e.g., Linux, Microsoft Windows, and FreeBSD), SCTP is rarely used. Especially, no realizations of iWARP over SCTP currently exist.

3.2 LibRIPC

LibRIPC, introduced 2012 in [45], is a network library designed for tightly interconnected cloud systems. It is message based and provides a neat interface, which can be adopted by applications without excessive effort. In order to support various types of use cases, [45] identifies two traffic classes and provides one message type for each of these classes.

One of the identified classes represents distributed applications, for instance distributed databases such as MongoDB [1]. A distributed application performs a parallel workload and spreads multiple threads over several nodes in a network. All threads combined fulfill exactly one task and complement each other. Characteristic messages for these applications tend to be small, as they primarily include information in order to synchronize the threads. These synchronization messages may occur frequently, may have high demands on their latency, or both. In any way, the related overhead has to be kept as small as possible, either to reduce the overall overhead of a large number of messages, or to reduce the latency of each individual one.

The other identified class can be summarized as client-server systems, such as http or mail servers. In this case, communication partners are different processes, one of which provides a service, which the other uses. Typically, a small number of servers provide their service for a large number of clients. Messages are likely to be large, for instance files stored on a ftp server. The latency of these messages does not tend to be critical, but a high bandwidth is necessary to provide a fast data transfer.

In order to address both of these traffic classes, LibRIPC provides short and long messages. The first is designed for small but latency critical messages, the latter for large pieces of data, which depend on a high bandwidth.

Beyond the support of these traffic classes, LibRIPC aims at several other goals. In the next Section we discuss these additional goals and defer to a deeper look on the message types. Following this, we present an overview of LibRIPC's interface.

3.2.1 Design Goals

LibRIPC was designed to match several goals, namely a small interface for ease of integration, message-based communication supporting well performing bulk data transfer and command messages, good portability to other hardware architectures and good suiting to cloud computing. We consider bulk data transfer as well performing, if the transfer provides atomicity as well as high throughput and reasonable low latency. In this context, atomicity means that a message does either arrive completely and without data corruption, or not at all. We consider

command messages as well performing, if messages are transferred with low latency. A further requirement for good cloud-computing suitability is resilience to the migration of processes to other machines/network addresses. In the following we describe these goals in more depth and introduce concepts that are used in LibRIPC to achieve them.

Portability to further network fabrics is an important factor, as it is a goal of LibRIPC to make applications work on various different network architectures. That means, an application that uses LibRIPC should require no modifications, if the underlying network fabric changes. The interface therefore hides every specifics of a network architecture and is designed so that every fabric can be narrowed down to it. Parameters of the interface are kept generic, raw C-pointers are used to specify memory regions and no network specific addressing schemes are used. Instead, LibRIPC provides its own address labels, called service IDs. The translation of these service IDs to the network address of their hosts is transparent to the application.

Ease of integration means, that existing applications that currently use another network library, such as the socket API, can be ported to LibRIPC with reasonable small effort. This factor is important, because there already exist a lot of applications for cloud computing. In order to profit from LibRIPC, these applications must be modified to use LibRIPC's interface. The necessary modifications should be kept within certain bounds to reduce the deterrence of the effort. This certainly includes ease of use in the development of new software from scratch.

LibRIPC's concept to provide ease of integration is a relatively small interface. Applications do not need to perform any kind of configuration before the communication can start. The allocation of a service ID is essentially the only action an application must take, before it can begin to send and receive messages.

Cloud-computing adequacy means, that LibRIPC has to fit in the major use cases which can be find in cloud environments. This includes applications using both of the communication classes mentioned earlier.

Another property of the cloud environment is that processes may be migrated to another machine, and therefore to another hardware address. Possible reasons for such process migration may be load balancing or maintenance purposes. LibRIPC has to be resilient against these address changes. This is a further benefit of the service IDs, which we already identified as a mechanism to provide the abstraction of the network fabrics. Service IDs identify users of the library and are translated to the corresponding host's network address. When a service intends to pass a message to another service, the correct network address is resolved. If this

address changes throughout the communication, LibRIPC resolves the new host of the service ID and is then able to continue the communication. This process is totally transparent to the application.

Message passing with support of both well performing bulk data-transfer and small messages is the key concern of LibRIPC. Naturally, the satisfaction of this demand highly depends on the network architecture, as software can only max out the performance provided by the hardware. Nevertheless, it is the responsibility of LibRIPC to use concepts which reach the given performance maximum.

In order to achieve low latency, it is necessary to minimize the protocol overhead. For instance, the current InfiniBand based implementation uses an unreliable datagram transfer, which requires no precedent buffer advertisement and yet provides RDMA. The size of short messages is limited, and it is therefore possible to provide a target memory-region in advance, which is large enough to contain any short message that may arrive. LibRIPC does not specify reliability for short messages, so that this datagram transfer is valid, although a loss of messages may occur. The approach of unreliable short messages is taken, because the implementation of reliability would add overhead to the data transfer, because at least one acknowledgement packet would have to be send, to indicate message arrival. Additionally, reliable transfer often requires a foregoing connection establishment. In order to prevail atomicity of short messages, their size is limited to the network's maximum transfer unit (MTU), so that a short message can always be send in one packet.

For long messages, protocol overhead can be accepted to a certain extent, as the transfer of large pieces of data is dominated by the network throughput. Therefore, the priority is to use mechanisms which allow for high throughput, such as CPU offloading an zero-copy data transfer. Atomicity for long messages requires the transmission to be reliable, so that lost packets can be detected and resend, and the data integrity is guaranteed. Often, reliability requires connection oriented communication, so that a connection establishment is necessary and adds to the overhead. Furthermore, the varying sizes of long messages require it to provide individual memory regions, which in turn make a proper buffer advertisement necessary. However, both connection establishment and buffer advertisement have a constant overhead, that is, the overhead occurs exactly once per message, no matter how large this message is. The general performance of the transfer is not affected and with increasing data length, the overhead becomes more and more negligible.

A further concept to limit the overhead of data transfer is used in the InfiniBand specific implementation of LibRIPC. As described, a registration of memory regions must precede any RDMA transfer, either dynamically for each individual

message (if the length of the message varies), or statically for an arbitrary message, which may arrive (if the maximum length of the message is known beforehand). Usually, this registration implies significant overhead [29]. In order to reduce this overhead, LibRIPC maintains memory regions, which are already registered but no longer used by the application. If the application needs to allocate a memory region, chances are that there already exist one that is large enough and may be re-used. In this case, the registration overhead is saved. Similar concepts can, but need not be used by implementations for other network architectures.

3.2.2 Interface

Portability was one of the design goals for LibRIPC. The interface abstracts from any specifics of the underlying network architecture, and allows it for any application using LibRIPC, to run over any network architecture that LibRIPC supports. In order to achieve this abstraction, our porting needs to implement the interface.

We describe this interface in the following paragraphs. For this purpose, we divided the interface into three parts: Methods for the allocation of service IDs, methods for the buffer management, and methods for the passing of messages.

Service ID Allocation

The service ID is the address scheme of LibRIPC. Every application that wants to communicate with others via LibRIPC needs an own service ID and needs to know the service ID of its communication partner, in order to send messages to it. LibRIPC provides to different ways for the applications to allocate a service ID. The application can either register a specific service ID, or let the library generate an arbitrary one.

In real-world scenarios, at least one application needs to acquire a specific service ID, so that its communication partners can initiate the communication. For instance, in a client-server model, the server would have to allocate a specific service ID, which all clients know. The clients, in turn, can allocate an arbitrary service ID and then initiate the communication with the server, who gets to know the clients service ID upon reception of its message and can answer to it.

If one of the allocation methods is called, LibRIPC also initializes itself, that is, it binds itself to a network adapter and starts the resolver and potential other subsystems. However, the initialization can also be called directly from the application, in which case it is skipped in following service ID allocations.

Buffer Management

In several high performance network architectures, the involved memory must be registered with the network adapter, in order to perform the transmission. LibRIPC provides a flexible buffer management in order to perform this registration, so that the application does not need to know the specific mechanisms and possible constraints of the network architecture. The application has three different possibilities, to allocate a buffer registered with the network card, which can therefore serve as the source or the sink of a transmission. A buffer can be

- ... newly allocated, that is, LibRIPC allocates a new memory buffer and registers it with the network card, or
- ... registered with the hardware, that is, LibRIPC is passed an already allocated buffer and performs the network specific registration process, or
- ... taken from LibRIPC's pool of memory regions, that is, LibRIPC re-uses a memory region, or allocates a new one, if no fitting memory region is available.

Furthermore, LibRIPC provides a method to free a buffer. That is, the buffer is added to the pool of available memory regions, and will be re-used the next time when an application requests memory.

Message Passing

The message passing interface of LibRIPC consists of one method to receive messages and one send method for each of both message types. When an application performs a receive call, it neither has to specify from which service ID the messages shall originate, nor the type of the message it expects. Instead, LibRIPC fills this information in variables that the application passes to it.

In order to send messages, the application must create an array of pointers to the messages, an array containing their sizes, and the number of messages the array contains. This information must be passed to the according method, either the one for short or for long messages. Additionally, both message types accept a list of return buffers. In this list, the application can store pointers to buffers, which LibRIPC can use as targets for RDMA transfer. The according information (i.e., the STags) are passed to the remote service. If the remote service issues long messages, it performs RDMA write operations into these buffers.

3.3 Approach

In this section we describe our approach for the integration of iWARP into LibRIPC. We aim at the best performance possible under the condition of full conformity to LibRIPC's interface. In order to describe our approach we divide the

interface into the same three subsystems as in the previous section: Allocation of service IDs, allocation of memory buffers and sending and receiving of messages.

Service IDs are an abstraction of the addressing scheme of the network architecture. iWARP uses the Internet protocol family and therefore IP addresses. Allocation of service IDs can be adopted from the existing implementation, as the allocation algorithm is independent from any hardware specifics. However, we need a resolver to translate service IDs into IP addresses.

Memory buffers have to be registered with the network adapter to allow RDMA transfers. The network architecture may have certain prerequisites, which must be fulfilled, or properties, which must be considered. For instance, InfiniBand either handles memory regions as a list of pages, or as a list of memory blocks, which need not be aligned to pages but must be of the same size [39].

Many aspects have to be considered for message passing and receiving. For both of the two message types provided by LibRIPC we have to choose a suitable transfer mechanism, which the hardware provides. Additional aspects are establishment and management of connections between communication partners.

In the following, we discuss these three subsystems. First, we provide a general overview of the approach, in which we introduce the libraries we use and describe a very simple workflow that can arise when LibRIPC is used. Afterwards, we discuss the identified subsystems individually and in more depth. Thereby, we begin with the service IDs, including the resolver, proceed with the buffer management and finish the Section with a discussion of our messaging system.

3.3.1 Design Overview

Figure 3.4 shows an overview of the subsystems within LibRIPC and which external libraries they use. The buffer management uses the *ibverbs* library, which provides a syntactical interface of both the InfiniBand [39] (hence its name) and the RDMAP [35] verbs. The ibverbs library also provides the RDMAP semantics used by the messaging system. As iWARP uses TCP, communication partners always need to create a connection before the data transfer can start. The *rdmacm* library provides semantics for TCP/IP-based connection management in conjunction with ibverbs [76]. Dependent on the use case of LibRIPC, the messaging system uses UDP sockets, too. The socket API is further used for the resolver.

The libraries ibverbs and rdmacm are part of the *OpenFabrics Enterprise Distribution* (OFED) [3], a collection of libraries and drivers assembled by the OpenFabrics Alliance (OFA). The OFA is a joint venture of several vendors of network hardware, computer systems and software, such as Mellanox, Cisco, Intel, IBM, and Microsoft. Originally founded as the OpenIB Alliance, the primary goal was to develop an open-source software package for InfiniBand support. As of 2006, the OFA added interoperability to its agenda, starting with the integra-

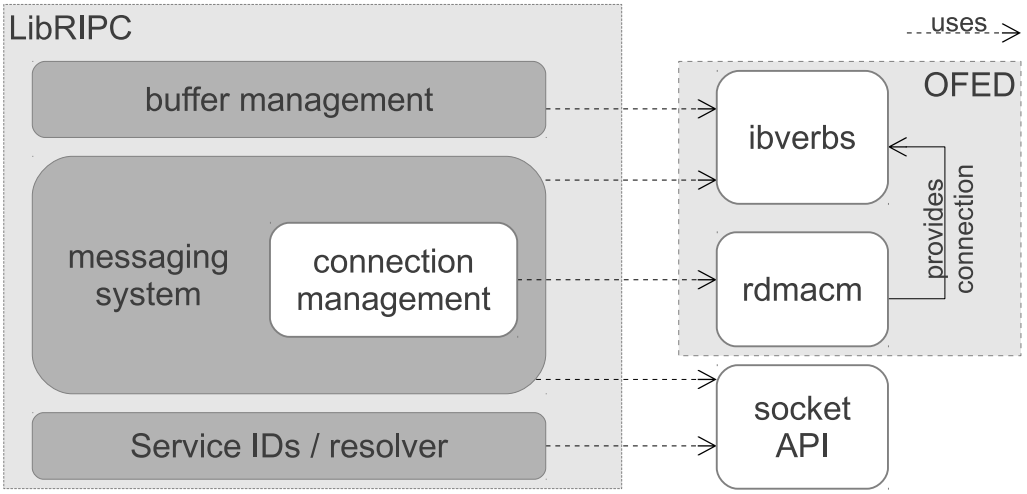


Figure 3.4: An overview of our design, depicting the used libraries and the tasks of LibRIPC. These tasks are the service IDs / resolver management, the buffer management, and the messaging systems, which contains the connection management. External libraries are the ibverbs which provide memory regions and RDMA semantics, rdmacm, which establishes connections over which the ibverbs perform its data transfer, and the socket API, which is used by the resolver, short messages, and control messages.

tion of iWARP as a supported fabric. The resemblance of the InfiniBand and the RDMAP verbs allows it to handle both architectures with the same library.

Using our approach may result in the following workflow when long messages are send: After the user application has registered or requested a service ID, it uses the buffer management of LibRIPC to allocate memory in which it can store the data that it intends to send. The application then sends a long message, and thereby passes the buffers and the service ID of the receiver to our library. LibRIPC resolves this service ID using UDP sockets, to get the IP address of the receiver's host. It then uses rdmacm to establish a connection to the resolved machine. Rdmacm binds this connection to the respective RNIC and provides the queue pair, which serves as the endpoint of the connection. The ibverbs library is then used to transfer the data to the remote machine. On the remote machine, the service with the addressed IDs performs a call to LibRIPC's receive method, and receives pointers to the messages, the number of received messages and the service ID of the sender.

The ibverbs library can be used with several different RNICs, because the OFED uses a dynamic library stack. For every RNIC the respective driver registers a set of functions, which the OFED library stack then uses. These functions

provide all the functionality needed to use the respective RNIC. This way, the ibverbs can be used for any RNIC, which supports this concept. Fortunately, many vendors of RNICs are members of the OFA. Their hardware is supported by the drivers integrated in the OFED [3] and therefore compatible to our approach.

In case of short messages, the workflow may be different. As already mentioned, our messaging system may use either ibverbs or UDP sockets for short and control messages. If UDP sockets are used, short messages require no connection establishment at all.

3.3.2 Service IDs and Resolver

As already mentioned at the beginning of Section 3.3, we can use the existing implementation of the allocation algorithm for service IDs. However, as service IDs only abstract from network addresses, we need a system to resolve the network address, which corresponds to a service ID. In the following, we describe our approach for this resolver.

Our resolver works very similar to the one in the InfiniBand-specific implementation of LibRIPC, which in turn is very similar to the Address Resolution Protocol (ARP) [60]. When an instance resolves a service, it sends a message to all other instances, containing its own address, its own service ID and the service ID it wants to resolve. This message is called *resolver request*. All instances that receive a resolver request save the according information (i.e., the address of the sending service) regardless of whether they currently need the information or not. If the querying service is addressed in the future, this circumvents an additional resolution process. The instance that hosts the requested service ID additionally sends an answer to the querying instance, providing its address; this message is called a *resolver reply*. To make sure that every running instance of LibRIPC receives a resolver request, we either need multicasting (all instances join one multicast group) or broadcasting (every machine in the network receives the resolver request). Multicasting would be more desirable, as broadcast messages flood the whole – possibly very large – network, even if only a small subset of the nodes run LibRIPC.

Although multicasting was identified by the OFA as a potential future feature for iWARP [49] and is planed to become part of the RDMAP verbs [67], it is currently not specified. We therefore need an alternative and decided to implement the resolver using broadcast messages of the User Datagram Protocol (UDP) [61]. UDP is part of the Internet protocol family, therefore compatible to IP and provides an unreliable and connectionless data transfer. As UDP is implemented via the standard socket API, it suffers from the socket's disadvantages, as discussed in Chapter 2. However, the protocol mechanisms are narrowed down to the essentials, datagrams are just enhanced by the port numbers, the length of the data,

and an optional checksum. Despite the sending, no protocol mechanisms are necessary, so that no further resources are consumed. Therefore, we expect small latencies, as only few CPU cycles are spend on protocol processing, and the small resolver messages do not cause a lot of copying overhead. Thus, we believe that UDP is a suitable protocol for this purpose.

This approach aims at simplicity and low overhead, but suffers from the weakness, that the broadcast messages are transferred to the whole network. For the time being, we use this approach, nevertheless. As stated in [45], the resolver system is planned to be substituted by a more sophisticated system, such as Chubby [9], so that these shortcomings can be accepted for the time being.

3.3.3 Messaging System

The messaging system of LibRIPC supports two different types of messages, which put different requirements on the data transmission. Long messages require a reliable transfer, which reaches high bandwidths. Short messages need not be reliably transferred, but are supposed to be send with lowest latency possible.

As mentioned above, we use the ibverbs library to perform RDMA transmissions. This is possible because of the similarity of the InfiniBand and the RDMAP verbs, which the OFA used to merge the specifications into one library.

In order to perform RDMA transfer via iWARP, it is necessary to establish a connection to the communication partner, before the transmission can begin. We discuss our connection management in the following section and then address the message types, starting with short messages.

Connection Management

For connection management purposes, we use the rdmacm library, which is part of the OFED. Rdmacm provides all necessary mechanisms to establish a connection over RDMA-capable fabrics, and the subsequent usage of the ibverbs library via that connection. A connection identifier is created by rdmacm, which can then be used to access the context of the network device and to create the necessary queue pair for the RDMA semantics. For the connection establishment, a dedicated thread constantly listens on a TCP port for incoming connection requests. The used port is advertised via the resolving process. After the connection is established, the connection identifier is used to create a queue pair. This queue pair is bound to the network device by rdmacm. As already mention in Section 3.3.1, functions are registered for each device by the respective driver. The registered functions are then used by the ibverbs library.

Short Messages

As discussed in 3.2.1, short messages lay the focus on keeping the latency as small as possible. The latency is primarily influenced by three factors: packet processing overhead, the bitrate, and the network's latency. Packet processing is any work that is necessary before the data can be put on the wire. That includes header computation for protocols and possible overhead to make the data accessible by the network adapter. The bitrate specifies the time needed to put a certain amount of bytes on the wire. Network latency is the time that the message spends in the network. That is on the wire, in switches, routers or other network equipment. The bitrate and the network latency can not be influenced by our library. They depend on the used hardware and the network infrastructure. However, we can influence the packet processing overhead.

Because of protocol offloading and kernel bypassing, iWARP is capable of providing low latency [69]. Although it is not fully competitive to InfiniBand [25], iWARP outperforms Ethernet without RDMA [21, 64].

The interface of LibRIPC puts two requirements on our design of short messages. Firstly, LibRIPC allows it to pass multiple messages from different buffers and memory regions with one call to the library. That requires it to realize a data transfer in scatter-gather fashion in order to avoid multiple network packets. Secondly, the receiver does not have to specify the sender, when issuing the call to the receive operation. Consequences of this requirement is that the receiver must expect a message from any remote service.

RDMAP provides an atomic send operation, which maps well to the first requirement. It allows for scatter-gather transfer into a pre-defined buffer at the communication partner. The buffer does not need to be advertised beforehand, but must be large enough to contain the full message. As LibRIPC limits the overall message size to the networks MTU [45], the maximum size is known, so that this property raises no problem.

The second requirement presents us with a bigger problem. While InfiniBand allows it to configure a queue pair to receive messages from any source [39], iWARP binds a queue pair to one specific connection. It is therefore not possible to configure a queue pair to receive messages from arbitrary senders. Instead it would be necessary to check for arrived messages on every existing connection. In the context of tightly interconnected applications which are distributed over many nodes, this can lead to a significant increase of queue pairs, on which messages may arrive. To detect the queue pair on which a message is received, it requires to successively poll every receive queue. In cases in which the receive call is issued before the communication partner sends its message, this has the potential to waste many CPU cycles.

An alternative for RDMA atomic send operations is to use UDP, as already

done for the resolver. As pointed out in Section 3.3.2, UDP provides reasonable small overhead, keeping the packet processing overhead low. UDP also allows it to receive a message without specifying the sender beforehand, and is connectionless, whereas the atomic send operation of iWARP requires a preceding connection establishment. Furthermore, a call to UDP's receive function is blocking; while waiting for the message, the CPU can be used for other computations or can be kept idle to save energy. However, the usage of UDP is likely to introduce a higher latency. The socket API consists of system calls, introducing context switches, and the message is copied from user space to kernel space. Additionally, UDP does not provide a scatter-gather functionality. Admittedly, the `sendmsg` system-call allows it to specify multiple source buffers, but the receiver must pass the exact number of buffers and their sizes to the `recvmsg` system-call, in order to receive the messages correctly. In general, the receiver does not know these details. We therefore have to copy all messages into one buffer before the transmission can be initiated, introducing further overhead.

Comparing the benefits and drawbacks of these two approaches, we came to the conclusion that the better approach varies from use case to use case. Applications without need of long messages can profit from UDP, as it saves the overhead for the connection establishment. But if many short messages are passed, for instance because of a long run time or a need of much synchronization, the benefits of the lower latencies of RDMA may amortize this overhead, as the connection is established only once per communication partner. The blocking behaviour of UDP can be disadvantageous and beneficial, too, depending on the use case. Applications that time their communication very well, so that the time spend on polling the receive queue is smaller than the overhead introduced by context switches of the blocking UDP, would perform better with RDMA atomic sends. If send operations are rather arbitrary (for instance in a client-server model, where the server does not know when a client may request a service), a blocking receive can save significant CPU time.

Because of these points, it is not possible to make a universally valid statement of what the better option is. We therefore realized both alternatives, and leave the choice up to the user of our library, who is capable of judging its application.

Long Messages

Long messages are designed to provide fast transfer for large data. In this context, large means every payload which is not transferable with one network packet, due to the network's MTU. Requirements on long messages are good performance, that is, high throughput and acceptable latency, and integrity of the data. To assure data integrity, a reliable data transfer is need. Fortunately, TCP provides a reliable service, so that data integrity is always given when using iWARP, so that we do

not need to keep special attention on that part.

For the sending of variably sized messages, the RDMAP verbs provide RDMA write and RDMA read operations, as discussed in Section 3.1. In order to use these operations, both the sender and the receiver have to register a memory region large enough to store the data. One of the involved communication partners plays the active role in the transmission, while the other's role is passive. When the RDMA write operation is used, the sender plays the active role and writes its data directly into the receiver's memory region. Inversely, the receiver plays the active role within a RDMA read operation. In this case, the data is read from the sender's memory region. Obviously, the active part of the transaction must know the correct memory region of its communication partner. The action of providing this information is called "buffer advertisement". LibRIPC allows buffer advertisement to be transferred piggyback, with both short and long messages.

The sending process of long messages uses both types of RDMA operations, and is split into three parts. In the first step, only RDMA write operations are performed. As described in Section 3.2.1, each instance of LibRIPC holds a list of known memory regions of its communication partners. For every message that is to be send, the list is searched for a memory region of the receiver, which is large enough to store the message. If such a memory region is found, the RDMA send operation is initiated and the data gets transferred. In the second step, an informational message is passed to the receiver. This message contains information of all the messages. Already transferred messages are marked as such, and for the messages which still are to be transferred, the buffer advertisement is done. The third step is then performed at the receiver, who performs a RDMA read operation for all remaining messages.

Step two is necessary in every case, even if all messages can be transferred via RDMA send operations. As the sender plays the active role, the receiver does not notice anything of the transmission. The informational message is therefore necessary, to inform the receiver of the messages.

To transfer the informational message, the same mechanism as the one for short messages is used. This reduces the overhead at the receiving end, as both short and long messages can be handled identically. As the short messages were designed for a low latency, this also takes care of keeping the protocol overhead for long messages reasonably low.

Chapter 4

Evaluation

In this chapter we evaluate our approach of porting LibRIPC to iWARP. We first state the goals of our evaluation in Section 4.1. In Section 4.2, we describe details of how we implemented our approach. Next, we discuss conducted experiments and their results in Section 4.3, and finish the chapter with a summary.

4.1 Evaluation Goals

Our evaluation aims at two main goals. The first is to assure that we succeeded with the porting itself. That is, that applications which were written to use the InfiniBand prototype of LibRIPC do not have to be modified in any way, in order to work with our implementation. This is one of the main goals, because the interface compliance of our porting was an explicit goal of our approach, as stated in Section 3.3.

The second main goal is to measure the performance that our implementation is capable to provide. Besides the interface compliance, high performance was the second goal of our approach. We therefore conducted performance tests, and compared the results with alternative solutions for network communication.

As LibRIPC provides two message types that target different patterns of network communication, we conducted tests for two parameters. One of these parameters is the throughput, which is the key concern for long messages. For short messages, the key concern was the latency of the communication. In order to evaluate the latency of our approach, we distinguished between the two different approaches, which we discussed in Section 3.3.3.

Besides these two goals, our evaluation yields one further important output. In our experiments we use applications that were already used with the InfiniBand based prototype implementation of LibRIPC. With their usage, we are capable of judging the initial design of LibRIPC in terms of its portability. Portability

was named one of LibRIPC's design goals in its introduction thesis [45]. As we performed the first porting of LibRIPC we gained the first insights, whether this goal was achieved or not.

4.2 Implementation of the iWARP Subsystem

With our porting of LibRIPC to iWARP, we confirmed the high level of portability of the design of LibRIPC. During our implementation we neither needed to compromise with, nor change the interface at all. Effectively, the prototype implementation of LibRIPC needed change in only 26 lines of code shared by all architecture specific implementations. These lines were exclusively architecture specific members of structs, which just needed to be adopted to the InfiniBand-specific code. Nevertheless, the portability can not yet be regarded as proven, as iWARP and InfiniBand share a significant amount of concepts, such as queue pairs and memory regions. A porting of LibRIPC to a less similar network architecture, such as Blue Gene Networking [12, 30], would be needed to further test the portability, and might be a subject for future work.

In Section 3.3 we described the design of our porting, which subsystems we use and how they interact with each other. The following sections reveals several details of how we implemented these concepts.

Service IDs and Resolver

We did not change the allocation algorithms of service IDs by any means, as they are completely unrelated to any architectural specifics. The resolver is implemented using UDP sockets and resides in an own thread. This thread is almost always in a blocking state, as it constantly waits for a resolver request to arrive. When this happens, the thread saves the included information and answers if its instance of LibRIPC hosts the demanded service ID.

A resolving process is initiated in case of a send call to a service, which the running instance of LibRIPC does not know, yet. It is therefore handled in the context of this library call and armored with a timeout mechanism, in case the requested service ID is currently unassigned.

Besides the requested service ID, and the ID of the requesting service, a resolver request contains the IP address of the sending instance, and several port numbers. These port numbers include the port on which the resolver reply is expected, the port on which RDMA connection requests are expected, and, in case of UDP based short messages, the port on which these messages are received.

Buffer Management

The buffer management is completely adopted from the implementation for InfiniBand. This is possible, because InfiniBand and RDMAP specify memory regions very similar. As we furthermore use the same library as for InfiniBand, no modifications were necessary. Details of the buffer management can be found in [45].

Connection Management

Similar to the resolver system, the connection management system maintains a dedicated thread, which constantly waits for incoming connection requests, and executes the initiation of a connection in the context of the sender. The rdmacm library performs a handshake, in which all necessary information is exchanged. First, the initiator of the message creates a queue pair and then issues a connection request, which raises an event at the remote thread. A connection request carries a payload containing the ID of the requesting service and of the service with which the connection shall be established. When a valid connection requests arrives, that is, the requested ID is hosted by the receiver of the request, the remote thread creates its own queue pair. It then sends a connection acceptance, which in turn raises an "connection established" event at the initiator. As soon as this event is received by the initiator, the remote thread receives the same event and the rdmacm connection is established. The created queue pairs are now ready to be used by the messaging system.

Short Messages with UDP

The sending process of short messages via UDP is divided into two steps. First, we have to emulate the scatter-gather mechanism, which UDP does not provide. For this purpose, a new buffer is allocated and every message is copied into this buffer, aggregating all messages. This buffer is then send via UDP and freed afterwards.

A slightly more complex part of the sending process takes place at the receiving end. In real life applications we have no guarantee that a receive call is issued before the sender sent the message. If a UDP message arrives before the corresponding receive call is issued, this message gets lost. We therefore implemented a dedicated thread, which constantly listens for incoming UDP messages. Every service ID has an associated list of message buffers, in which incoming messages for this service are inserted. When the application issues the receive call, the first element in the list is taken out and handed back to the application. If no message has arrived yet, the application is blocked, and is woken by the message receiver thread when a message arrives.

Short Messages with RDMAP

As opposed to short messages via UDP, a connection establishment is required to send short messages via RDMAP. For every connection that an instance of LibRIPC holds, a receive work-request is posted to the corresponding receive queue. When a short message is sent, an atomic read operation is issued, which places the message directly in the buffer of this work request at the remote end. Unfortunately, the receiver has no way of knowing, on which of the connections a message may arrive. We therefore maintain a list of all receive queues, which is polled when an application issues a receive call. As soon as a message arrives on one of the receive queues, a new work request is posted for the corresponding connection, and the received message is passed to the calling application.

Long Messages

We implemented long messages straight forward based on the description in Section 3.3.3. Every message for which a suitable memory region is available is passed via a RDMA write operation. A message containing all necessary information about the transmission is then passed to the receiver using the same mechanism that is used for short messages. This message contains the address of the long messages (either where they were written via RDMA write, or from were they can be read), the sizes of the messages, the STag (in case the receiver must fetch the message via RDMA read) and a flag that signals whether the message was already transferred via RDMA write, or not. All messages that could not be transferred, yet, are then fetched by the communication partner via RDMA read operations.

4.3 Experiments

In order to evaluate our porting, we conducted several experiments to measure how our approach accomplished its objectives. Basically, we applied the same performance tests that were already used in the introduction of LibRIPC in [45]. This approach has several advantages. On the one hand, this saves time, as the used applications are already available. On the other hand, we are able to compare our results with the ones of the InfiniBand based implementation. Furthermore, we are able to proof the interface compliance, if the used applications require no modifications. Aside from that, these tests were well-considered, and as our implementation targets the same goals as the initial implementation, the tests are well suited to evaluate our porting.

Unfortunately, we lack iWARP-capable RNICs to run our experiments on. We therefore conduct them by using the Softiwarp implementation of [79] (see Sec-

tion 2.2) on top of InfiniBand network adapters with IP over InfiniBand (IPoIB). The result of this setup is, that we are not able to profit of all of iWARP's mechanisms. For instance, we have no protocol offloading, as IPoIB is a software library and not a functionality of the InfiniBand adapter itself. As opposed to that, the zero-copy mechanisms are still applied, as they are supported by both Softiwarp and IPoIB. Kernel bypass is only partially achieved by our setup. All packets are processed by the TCP/IP stack of IPoIB, which resides in the kernel. However, the frequent protection domain crossings that we experience with the socket API are avoided, as the protocol processing is done in kernel, by the Softiwarp module.

Nevertheless, we believe that the results of our experiments are meaningful. Using Softiwarp does not affect our first evaluation goal, the interface compliance. Softiwarp can be used with the OFED [54] and therefore needs no modifications of our code. We used this implementation of software based iWARP, as it is the only one that is compliant to the OFED and is capable of achieving performance improvements. Additionally, it is the only project, which is still worked on.

The second evaluation goal – performance measurement – obviously is affected by using Softiwarp. Because of the mentioned points, we can not expect our setup to perform as well as the usage of a RNIC capable of iWARP. However, we can expect significant performance improvements compared with the socket API, as Softiwarp uses the sockets completely in kernel space [58]. Furthermore, we are able to estimate, whether our results are heavily affected by the lack of an RNIC or not. We will include this knowledge in our interpretations and therefore gain a valid evaluation of our approach.

In order to evaluate the latency of our library, we used a simple ping-pong application, in which two communication partners repeatedly send its counterpart a message, wait for it to reply and then send the next message. We used this test with long messages as well as with both types of short messages and made use of payloads of different length. Despite long messages not being designed for low latency, we applied this test on long messages to evaluate, if the deployment of a designated message type for short messages is beneficial.

We conducted more thoroughly testing of long messages with the Jetty web server [28]. Jetty is written in Java and is a http server, which can host both static content and dynamic servlets. Besides serving as a stand-alone server, Jetty can be (and is [27]) embedded in other applications. We use Jetty as a stand-alone server, performing several transfers of files with varying sizes, which significantly exceed the given limit for short messages. Details of how LibRIPC was integrated in Jetty are discussed in [45].

In the following sections we describe the application of the planned experiments. We first describe the test system on which the experiments were conducted and then present and interpret the results.

4.3.1 Test System

To conduct our experiments, we used two nodes of a small four-node cluster. Each of the nodes is equipped with an Intel Xeon E5520 quadcore CPU clocked at 2.27 GHz and 6 GB of DDR2 RAM. As network interconnection we use the installed Mellanox ConnectX-2 QDR InfiniBand adapters with Softiwarp on top of IPoIB, as mentioned in 4.3. The adapters are connected via a HP InfiniBand DDR switch and provide an effective bandwidth of 16 GB/s. A CentOS 6.2 with a 64-bit Linux kernel version 2.6.32 is installed on both nodes. The Softiwarp software needed a slight modification in order to compile for the used kernel, and to create a virtual device over IPoIB. Overall, this changed four lines of code and did not affect any concepts or mechanisms of Softiwarp.

4.3.2 Results and Interpretation

In this section we present and discuss the results of our experiments, first the ping-pong test and then the Jetty test.

Ping-Pong Latency Test

In order to evaluate the latency of our implementation, we used the same ping-pong test as in [45]. A client repeatedly sends messages containing pseudo-random data to a server and waits for the server to reply, before the next messages is sent. The receiving server answers every incoming messages by replying with the exact same message. In one run, the client always sends the same message, which is generated before we start measuring the transfer time. We conducted the experiments with messages of a size between 50 and 1400 bytes, with 50-byte steps. For every size, the message was sent 10000 times between the two nodes. We recorded the time needed for all of the transfers and computed the average round trip time by diving the time by 10000.

The experiments were conducted with six different approaches. One being a socket based implementation using TCP, and four being LibRIPC with varying implementations and used message type. For both implementations – one using UDP, one using RDMAP for short and control messages – we tested both message types. The fifth approach is LibRIPC over InfiniBand.

Figure 4.1 shows the results for the ping-pong test. Our UDP based implementation of LibRIPC needed about half as much time as the sockets, regardless of which message type was used. Short messages needed from 27.28 microseconds (50 bytes) through to 32.12 microseconds (1400 bytes) and were only slightly faster then long messages, which range from 32.5 (50 bytes) through to 33,45 microseconds (1400 bytes). The difference between those two message types is

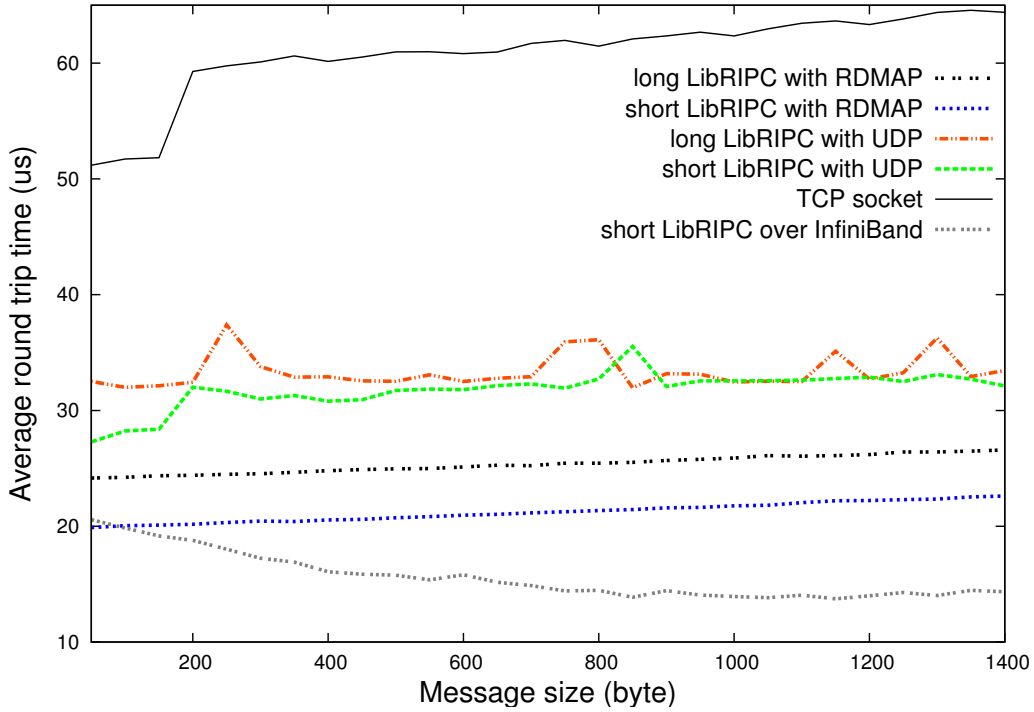


Figure 4.1: Results of the ping-pong test using TCP sockets and LibRIPC over both iWARP and InfiniBand. The figure shows the results for both implementations (i.e., with UDP and RDMAP based short messages) for both message types.

only marginally, despite the long messages effectively performing two transfers (the sending of the control message and the RDMA operation). We believe that this indicates a large impact of the system call that is performed for both short and long messages. Additionally, we suspect the system calls to be the reason for the outliers that can be seen for both of the UDP implementations. In three cases (850, 1000, and 1050 bytes), the long messages were even faster than the short messages.

The average round trip time of messages transferred by TCP sockets range from 51.28 microseconds for 50 bytes messages through to 64.39 microseconds for 1400 bytes messages. The degree of this difference to our UDP based approach is a little surprising, as the short messages of our implementation of LibRIPC also use the socket API, and add further overhead due to an extra copy process, as LibRIPC provides scatter-gather functionality. Protocol overhead added by TCP is the only possible reason that we can make out for this effect.

Of our implementations of LibRIPC over iWARP, the RDMAP based implementation achieved the best results. Here, the long messages needed 24.16 mi-

croseconds for 50 bytes messages through to 26.58 microseconds for 1400 bytes messages, and were only slightly slower than short messages that ranged between 19.89 (50 bytes) and 22.61 microseconds (1400 bytes). We promise ourselves even better results when a RNIC capable of iWARP is used. The TCP protocol overhead showed to be an important factor compared to UDP messages, therefore we believe that the protocol offloading would introduce a further performance improvement.

Despite the unexpectedly high latencies for small messages (20.57 microseconds for 50 byte), the InfiniBand based LibRIPC yielded the best results. Whereas the latency reached its maximum for the smallest message, the fastest transfer was achieved for 1150-byte messages (13.73 microseconds). The observed average round trip time for messages of 1400 bytes was 14.34 microsecond. However, the fact that the InfiniBand implementation reached the lowest latencies endorses our interpretation, that protocol offloading could improve the results of our iWARP based approach.

Jetty Web-Server Test

In our second experiment we lay focus on the performance of long messages. We used the web server Jetty [28], which runs on top of TCP, to evaluate the performance when sockets are used. Additionally, we used a modification of Jetty, which runs on top of LibRIPC and is introduced in [45]. For the usage of TCP, we implemented a simple client application that uses *libcurl* [72] to create http requests and to parse the responses. In order to test the performance of Jetty over LibRIPC, we used a second application, that passes http get-requests via short sends and receives the http responses and the file data via long sends. We conducted the experiment only with our UDP based implementation of LibRIPC, as both variations use the same RDMA operations. Because the transferred data is comparably large, there are no significant differences to expect in this experiment.

Both client applications request a file via http get and wait for the response and the data to arrive. As in our ping-pong tests, we performed 10000 repetitions for each file and computed the average transfer time. The files were created with sizes varying between 128 KB and 80 MB and are filled with pseudo-random data.

Figure 4.2 shows the results of this experiment. As with the ping-pong tests, LibRIPC over iWARP outperforms the transfer via TCP sockets by far. Using LibRIPC, the average transfer time ranged from 0.185 milliseconds for a 128 KB file through to 39.76 milliseconds for a 80 MB file. With TCP sockets used, the average transfer time was almost the double than with LibRIPC (0.36 milliseconds) for the smallest file (128 KB), and the treble (118.01 milliseconds) for the 80 MB file.

The results for the small files are not surprising, as they reflect the results from

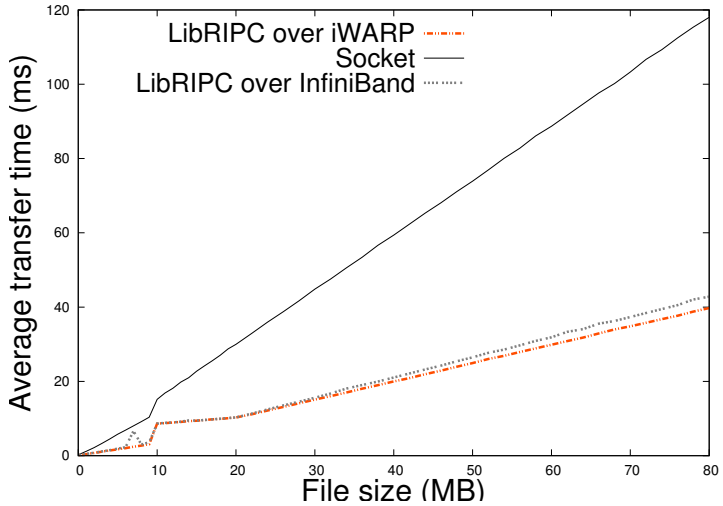


Figure 4.2: Results of the jetty web-server test using TCP sockets and LibRIPC over iWARP and InfiniBand.

our first test. However, for the larger files, the latency of the messages grows less important, as it gets dominated by the throughput. We can therefore conclude, that the data transfer benefits strongly from the use of RDMA.

As opposed to that, the protocol overhead does not seem to be of great significance in this test. This makes sense, as we run our experiments with no pressure on the CPU, so that the increasing CPU involvement due to protocol processing does not become a bottleneck.

LibRIPC over InfiniBand yields no performance improvements. To the contrary, with 0.19 milliseconds for 128 KB files and 42.82 milliseconds for 80 MB files, the InfiniBand implementation is constantly slower. We therefore expect no strong improvements when RNICs capable of iWARP are used in this scenario. However, we do believe that in situations of CPU pressure, our good results will significantly decrease, as the protocol processing will suffer under the resource contention with other processes.

4.4 Summary

In this evaluation we aimed to proof our approach to be functional and well performing. With the tests we conducted, we were able to show that the approach yields reasonable results. Despite the usage of software based iWARP, the performance of our implementation surpassed the socket API by far. Additionally, our approach outperformed LibRIPC over InfiniBand for long messages. We expect

that these results become even more distinct, when appropriate hardware is used. The good results of several performance tests [17, 21, 25, 62, 69] encourage us in this expectation.

Besides the good performance results, our implementation is completely compliant to the interface of LibRIPC as introduced in [45]. We could therefore show the validity of our approach, as well as the portability of LibRIPC.

Chapter 5

Conclusion

In current IT trends, the concept of cloud computing plays a very significant role. Several providers of clouds have emerged, such as Amazon with EC2 [4] and Google with its cloud platforms [38], which address business costumer that plan to outsource their IT departments, in order to save costs. We identified high demands on the networking infrastructure of these cloud systems, which require high performance in combination with reduced pressure on the CPU cores. However, current cloud applications often rely on the Berkeley socket API, which does not correspond to these demands.

The network library LibRIPC overcomes theses issues of sockets in terms of performance and CPU load. Its neat, message based interface eases the integration in both new and existing applications and makes no assumptions of the underlying hardware. However, the current prototype implementation for InfiniBand yielded good results, but requires relatively expensive InfiniBand compliant hardware.

With our porting of LibRIPC to iWARP, we combine the flexibility and ease of integration of LibRIPC with the cost efficiency of Ethernet without abandoning the performance benefits of InfiniBand. With an experimental evaluation we were able to verify this claim and achieved promising results. Despite running the tests over software based iWARP, our approach outperformed TCP sockets by far in terms of both latency and throughput. In comparison with the existing InfiniBand based prototype, our implementation added little latency, which we believe to be avoidable with the use of hardware based iWARP.

5.1 Future Work

Despite our successful porting of LibRIPC to iWARP, we could not ascertain the portability of LibRIPC beyond doubt. For this purpose, iWARP's resemblance to InfiniBand is to large. It is therefore an explicit goal to enable LibRIPC for less

similar network architectures as well. One promising goal is the Blue Gene supercomputer [12, 30] as the interconnection of Blue Gene is completely dissimilar to InfiniBand and iWARP.

Another field of work is the resolver of LibRIPC, which is currently kept very simple for both implementations, and which we do not believe to scale well, because of the usage of multicast and broadcast, respectively. Especially for our implementation for iWARP, this is of special importance. Not only is the flooding of the network intensified with the usage of broadcasting, compared to multicasting, the UDP based approach introduces another problem. To date, all instances of LibRIPC listen on the same UDP port, to which all resolver requests are send. Unfortunately, for multiple instances on the same host, it is not possible to listen simultaneously on the same port. As a consequence, our implementation is currently limited to one instance per host, because all but the instance that starts first are not able to bind to the port. It is therefore urgent for us to find and implement are more sophisticated solution.

In order to further verify our approach, it is necessary to evaluate our implementation on iWARP compliant hardware. Despite our promising results with software based iWARP, the performance improvements that we promise ourselves from iWARP RNICs are still to be proofed.

Finally, we want to conduct further experiments in order to discriminate the performance of UDP and RDMAP based short messages. In our latency tests discussed in Section 4.3.2, RDMAP was able to outperform UDP. But in the simulated scenario, the overhead of connection establishment was compensated by the high number of repetitions, as the connection needed to be established only once.

Bibliography

[1] 10gen Inc. MongoDB. `http://www.mongodb.org`.

[2] OpenFabrics Alliance. How iWARP Helps in HPC Cloud Computing Implementation, 2011.

[3] OpenFabrics Alliance. The Case for Open Source-RDMA and the OpenFabrics Alliance, August 2011. `https://www.openfabrics.org`.

[4] amazon web services. Amazon Elastic Compute Cloud (Amazon EC2). `http://www.amazon.com/ec2`.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009.

[6] Pavan Balaji, Hyun-Wook Jin, Karthikeyan Vaidyanathan, and Dhabaleswar K. Panda. Supporting iWARP Compatibility and Features for Regular Network Adapters. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, September 2005.

[7] Pavan Balaji, Hemal V. Shah, and D.K. Panda. Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. In *Proceedings of the 2004 Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies*, 2004.

[8] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *micro, IEEE*, 15:29–36, February 2005.

[9] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

[10] Inc. Calxeda. Calxeda EnergyCore ECX-1000 Series. Product Brief, May 2012. http://www.calxeda.com/wp-content/uploads/2012/06/ECX1000-Product-Brief-612.pdf.

[11] Don Cameron and Greg Regnier. *The Virtual Interface Architecture*. Intel Press, 1st edition, April 2002.

[12] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The IBM Blue Gene/Q Interconnection Fabric. *micro, IEEE*, 32(1):32–43, January 2012.

[13] Eunmi Choi, Bhaskar Prasad Rimal, and Ian Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, 2009.

[14] David D. Clark. The Design Philosophy of the DARPA Internet Protocols. *Computer Communication Review*, 18(4):106–114, August 1988.

[15] David D. Clark, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *Communications Magazine, IEEE*, 27(6):23–29, 1989.

[16] David Cohen, Thomas Talpey, Arkady Kanevsky, Uri Cummings, Michael Krause, Renato Recio, Diego Crupnicoff, Lloyd Dickman, and Paul Grun. Remote Direct Memory Access over the Converged Enhanced Ethernet Fabric: Evaluating the Options. In *Proceedings of the 17th IEEE Symposium on High Performance Interconnects*, April 2009.

[17] Chelsio Communications. iWARP – Robust, Proven Low Latency Ethernet Clustering. Whitepaper. http://www.moderntech.com.hk/sites/default/files/whitepaper/V09_iWAR_Summary_WP_0.pdf.

[18] Compaq Computer Corp., Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification*, 1.0 edition, December 1997.

[19] Intel Corporation. NetEffect Server Cluster Adapters. Product Brief, 2010. http://www.intel.com/content/dam/doc/product-brief/neteffect-server-cluster-adapter-brief.pdf.

[20] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. *Marker PDU Aligned Framing for TCP Specification*, October 2007. RFC 5044, http://tools.ietf.org/html/rfc5044.

[21] Dennis Dalessandro and Pete Wyckoff. A Performance Analysis of the Ammasso RDMA Enabled Ethernet Adapter and its iWARP API. In *Proceedings of the IEEE Cluster 2005 Conference, RAIT Workshop*, 2005.

[22] Dennis Dalessandro, Pete Wyckoff, and Ananth Devulapalli. Software implementation and testing of iWarp protocol. `https://www.osc.edu/research/network_file/projects/iwarp/iwarp_main`, last accessed 2012-09-20.

[23] Dennis Dalessandro, Pete Wyckoff, and Ananth Devulapalli. Design and Implementation of the iWarp Protocol in Software. In *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2005.

[24] Dennis Dalessandro, Pete Wyckoff, and Ananth Devulapalli. iWARP Protocol Kernel Space Software Implementation. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium 2006*, 2006.

[25] Dennis Dalessandro, Pete Wyckoff, and Gary Montry. Initial Performance Evaluation of the NetEffect 10 Gigabit iWARP Adapter. In *Proceedings of IEEE Cluster 2006*, 2006.

[26] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *Proceedings of the 13th Symposium on High-Performance Interconnects*, pages 58–63, 2005.

[27] Codehouse foundation. Jetty Powered. `http://docs.codehaus.org/display/JETTY/Jetty+Powered`.

[28] Codehouse foundation. Jetty Webserver. `http://jetty.codehaus.org/jetty/`.

[29] Philip Werner Frey. *Zero-Copy Network Communication: An Applicability Study of iWARP beyond Micro Benchmarks*. PhD thesis, ETH Zurich, 2010.

[30] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, Heidelberger P., Hoenicke D., Kopcsay G. V., T. A. Liebsch, Ohmacht M., B. D. Steinmacher-Burow, T. Takken, and Vranas P. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2-3):195–212, May 2005.

[31] Brice Goglin. Design and Implementation of Open-MX: High-Performance Message Passing over generic Ethernet hardware. *Journal of Parallel and Distributed Computing*, 37(2):85–100, February 2011.

[32] Robert L. Grossman. The Case for Cloud Computing. *IT Professional*, 11(2):23–27, March 2009.

[33] James Hamilton. WTIA: Scaling into the Cloud with Amazon Web Services, March 2009. `http://perspectives.mvdirona.com/2009/03/04/WTIAScalingIntoTheCloudWithAmazonWebServices.aspx`.

[34] Brian Hayes. Cloud Computing. *Communications of the ACM*, 51(7), July 2008.

[35] Jeff Hilland, Paul Culley, Jim Pinkerton, and Renato Recio. *RDMA Protocol Verbs Specification*, April 2003.

[36] David Hilley. Cloud Computing: A Taxonomy of Platform and Infrastructure-level Offerings. *Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-13*, April 2009.

[37] Urs Hölzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[38] Google Inc. Google Cloud Platform. `http://cloud.google.com`.

[39] InfiniBand Trade Association. *InfiniBand Architecture Specification*, June 2007.

[40] ISO/IEC. Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, November 1994.

[41] Hyun-Wook Jin, Pavan Balaji, Chuck Yoo, Jin-Young Choi, and Dhabaleswar K. Panda. Exploiting NIC architectural support for enhancing IP-based protocols on high-performance networks. *Journal of Parallel and Distributed Computing*, 65(11):1348–1365, November 2005.

[42] William Joy, Robert Fabry, Samuel Leffler, M. Kirk McKusick, and Michael Karels. Berkeley Software Architecture Manual 4.4BSD Edition. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1986.

[43] Inc. Juniper Networks. Delivering HPC Applications with Juniper Networks and Chelsio Communications. Whitepaper, August 2011. `http://www.chelsio.com/wp-content/uploads/2011/05/2000429-001-EN.pdf`.

[44] Humaira Kamal, Brad Penoff, and Alan Wagner. SCTP versus TCP for MPI. In *Supercomputing 2005*, November 2005.

[45] Jens Kehne. Light-Weight Remote Communication for High-Performance Cloud Networks. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, May 2012.

[46] Anne MacFarland. iWARP Wrings the Overhead Out of Ethernet. *The Clipper Group Explorer*, October 2004.

[47] Tudor Marian. *Operating Systems Abstractions for Software Packet Processing in Datacenters*. PhD thesis, Cornell University, Department of COmputer Science, August 2010.

[48] Tudor Marian, Ki-Suh Lee, and Hakim Weatherspoon. NetSlices: Scalable Multi-Core Packet Processing in User-Space. Technical report, Computing and Information Science (CIS) Department, Cornell University, July 2012.

[49] Felix Marti. iWARP 2.0. In *the 2010 OpenFabrics Alliance International Workshop in Sonoma, California*. OpenFabrics Alliance, March 2010.

[50] B. Metzler, F. Neeser, and P Frey. Softiwarp: A software iWARP driver for OpenFabrics. In *Proceedings of the OpenFabrics Sonoma Conference*, March 2009.

[51] Bernard Metzler. OpenRDMA Software Architecture, December 2004. `http://rdma.sourceforge.net/architecture.pdf`.

[52] Bernard Metzler, Philip Frey, and Animesh Trivedi. Softiwarp – Project Update. In *the 2010 OpenFabrics Alliance International Workshop in Sonoma, California*, March 2010.

[53] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of the 9th conference on Hot Topics in Operation Systems*, page 5, May 2003.

[54] F. D. Neeser, B. Metzler, and P. W. Frey. SoftRDMA: Implementing iWARP over TCP kernel sockets. *IBM Journal of Research and Development*, 54:5:1–5:16, February 2010.

[55] NetEffect. Understanding iWARP: Delivering Low Latency to Ethernet. Technology Brief. `http://www.intel.com/content/dam/doc/technology-brief/iwarp-brief.pdf`.

[56] Extreme Networks. Converged Enhanced Ethernet (CEE). Technical report, Extreme Networks, Inc., 2009.

[57] Inc. Netxen. NXB-10GXxR Intelligent NIC. Product Brief, 2007. http://www.sandirect.com/documents/netxen_NXB10GXxR.pdf.

[58] Foong Annie P., Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. TCP Performance Re-Visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, 2003.

[59] Gregory F. Pfister. An Introduction to the InfiniBand Architecture. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O*, chapter 42, pages 617–632. John Wiley & Sons Inc, 2001.

[60] David C. Plumer. An Ethernet Adress Resolution Protocol, November 1982. RFC 826, http://tools.ietf.org/html/rfc826.

[61] J. Postel. User Datagram Protocol, August 1980. RFC 768, http://www.ietf.org/rfc/rfc768.txt.

[62] Mohammad J. Rashti and Ahmad Afsahi. 10-Gigabit iWARP Ethernet: Comparative Performance Analysis with InfiniBand and Myrinet-10G. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium*, March 2007.

[63] Mohammad J. Rashti, Ryan E. Grant, Ahmad Afsahi, and Pavan Balaji. iWARP Redefined: Scalable Connectionless Communication over High-Speed Ethernet. In *HiPC'10*, pages 1–10, 2010.

[64] Casey B. Reardon, Alan D. George, and Clement T. Cole. Comparative Performance Analysis of RDMA-Enhanced Ethernet. In *workshop on High-Performance Interconnects for Distributed Computing*, 2005.

[65] A Romanow, J Mogul, T. Talpey, and S. Bailey. Remote Direct Memory Access (RDMA) over IP Problem Statement, December 2005. RFC 4297, http://tools.ietf.org/html/rfc4297.

[66] Hemal Shah, James Pinkerton, Renato Recio, and Culley Paul. *Direct Data Placement over Reliable Transports*, October 2002. RFC 5041, https://tools.ietf.org/html/rfc5041.

[67] Bob Sharp and Felix Marti. iWARP Enhancements - IETF Draft. In *the 2012 OpenFabrics Alliance International Workshop in Monterey, California*. OpenFabrics Alliance, March 2012.

[68] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of the SC01*, 2001.

[69] Tom Stachura. Cloud Networking with 10GbE & iWARP. In *the 2010 SC10 Trade Show in New Orleans, Louisina*. Intel Corporation, November 2010.

[70] Paul Stalvig. Introduction to the Stream Control Transmision Protocol (SCTP). whitepaper, October 2007.

[71] Matt Stansberry. 2011 Data Center Industry Survery. Uptime Institue, June 2009. `http://www.uptimeinstitute.com/publications`.

[72] Daniel Stenberg et al. libcurl – the multiprotocol file transfer library. `http://curl.haxx.se/libcurl/`.

[73] R. Stewart et al. Stream Control Transmission Protocol, September 2007. RFC 4960, `http://tools.ietf.org/html/rfc4960`.

[74] Jonathan Stone and Craig Partridge. When the CRC and TCP Checksum Disagree. In *Proceedings of the ACM SIGCOMM 2000*, pages 309–319, August 2000.

[75] Mike Tsai, Brad Penoff, and Alan Wagner. A Hybrid MPI Design using SCTP and iWARP. In *Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.

[76] Librdmacm Programmer's Manual. Linux manual page, July 2010.

[77] Tim S. Woodall, Galen M. Shipman, George Bosilca, and Arthur B. Maccabe. High Performance RDMA Protocols in HPC. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, September 2006.

[78] Eric Yeh, Herman Chao, Venu Mannem, Joe Gervais, and Bradley Booth. Introduction to TCP/IP Offload Enginge (TOE). Technology Paper, October 2002.

[79] IBM Research Zurich. Remote direct memory access. `http://www.zurich.ibm.com/sys/rdma/`, last accessed 2012-09-20.