Karlsruher Institut für Technologie

Light-Weight Remote Communication for High-Performance Cloud Networks

Diplomarbeit
von

cand. inform. Jens Kehne

an der Fakultät für Informatik

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

Bearbeitungszeit: 24. November 2011 – 23. Mai 2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 5. August 2014

Abstract

Over the last decade, customers increasingly tended to host their computing services in the cloud instead of buying and maintaining their own hardware in order to save costs. Due to cost- and power constraints, the data centers hosting these clouds are currently moving towards more balanced hardware designs than are currently in use. These new architectures employ “weaker” but more cost- and power-efficient CPU cores as well as interconnects capable of carrying out delegated work from the CPU independently by including features like transport layer offloading, user-level I/O and remote DMA.

Most current cloud platforms are based on commodity operating systems such as Linux or Microsoft Windows. While these operating systems can likely be ported to future clouds’ CPU architectures, their network stacks typically neglect support for offloading-, user-level I/O- or remote DMA-features that may be present in the network hardware. Especially the Berkeley socket interface used by most commodity operating systems is largely incompatible with these special features: Socket I/O is synchronous and stream-based, whereas most offloading mechanisms as well as remote DMA are asynchronous and message-based by nature.

In this thesis, we introduce LibRIPC, a light-weight communication library for cloud applications. LibRIPC improves the performance of cloud networking significantly, yet without sacrificing flexibility. Instead of using sockets, LibRIPC provides a message-based interface, consisting of a short- and a long send operation. Short sends provide low latency at the cost of a limited message size, while long sends use the hardware’s remote DMA features to provide high bandwidth. Both send functions implement zero-copy data transfer in order to reduce both latency and overhead. Hardware-specific details, like connection establishment or memory registration, are hidden behind library functions, providing applications with both ease of integration and portability. The library provides the flexibility needed by cloud applications by using a hardware- and location-agnostic addressing scheme to address communication endpoints. It resolves hardware specific addresses dynamically, thus supporting application restarts and migrations.

We evaluate our approach by means of an InfiniBand-based prototype implementation and the Jetty web server, which we integrated into the Hadoop map/reduce framework. Our prototype leverages InfiniBand's hardware semantics to implement efficient data transfer, while compensating InfiniBand's drawbacks by caching the results of expensive operations. We were able to efficiently integrate our library into Jetty; however, Hadoop would have required extensive modifications to its code base in order to make full use of our library's zero-copy semantics. Nonetheless, our results indicate that our library can significantly improve network throughput, latency and overhead.

Deutsche Zusammenfassung

Während der letzten 10 Jahre gewann das Cloud Computing immer weiter an Bedeutung. Um kosten zu sparen installieren immer mehr Anwender ihre Anwendungen in der Cloud, statt eigene Hardware zu kaufen und zu betreiben. Als Reaktion entstanden große Rechenzentren, die ihren Kunden Rechnerkapazität zum Betreiben eigener Anwendungen zu günstigen Preisen anbieten. Diese Rechenzentren verwenden momentan gewöhnliche Rechnerhardware, die zwar leistungsstark ist, aber hohe Anschaffungs- und Stromkosten verursacht. Aus diesem Grund werden momentan neue Hardwarearchitekturen mit schwächeren aber energieeffizienteren CPUs entwickelt. Wir glauben, dass in zukünftiger Cloudhardware außerdem Netzwerkhardware mit Zusatzfunktionen wie user-level I/O oder remote DMA zum Einsatz kommt, um die CPUs zu entlasten.

Aktuelle Cloud-Plattformen setzen meist bekannte Betriebssysteme wie Linux oder Microsoft Windows ein, um Kompatibilität mit existierender Software zu gewährleisten. Diese Betriebssysteme beinhalten oft keine Unterstützung für die speziellen Funktionen zukünftiger Netzwerkhardware. Stattdessen verwenden sie traditionell software-basierte Netzwerkstacks, die auf TCP/IP und dem Berkeley-Socket-Interface basieren. Besonders das Socket-Interface ist mit Funktionen wie remote DMA weitgehend inkompatibel, da seine Semantik auf Datenströmen basiert, während remote DMA-Anfragen sich eher wie in sich abgeschlossene Nachrichten verhalten.

In der vorliegenden Arbeit beschreiben wir LibRIPC, eine leichtgewichtige Kommunikationsbibliothek für Cloud-Anwendungen. LibRIPC verbessert die Leistung zukünftiger Netzwerkhardware signifikant, ohne dabei die von Anwendungen benötigte Flexibilität zu vernachlässigen. Anstatt Sockets bietet LibRIPC eine nachrichtenbasierte Schnittstelle an, zwei Funktionen zum senden von Daten implementiert: Eine Funktion für kurze Nachrichten, die auf niedrige Latenz optimiert ist, sowie eine Funktion für lange Nachrichten, die durch die Nutzung von remote DMA-Funktionalität hohe Datendurchsätze erreicht. Übertragene Daten werden weder beim Senden noch beim Empfangen kopiert, um die Übertragungslatenz zu minimieren. LibRIPC nutzt den vollen Funktionsumfang der Hardware aus, versteckt die Hardwarefunktionen aber gleichzeitig vor der Anwendung, um

die Hardwareunabhängigkeit der Anwendung zu gewährleisten. Um Flexibilität zu erreichen verwendet die Bibliothek ein eigenes Adressschema, dass sowohl von der verwendeten Hardware als auch von physischen Maschinen unabhängig ist. Hardwareabhängige Adressen werden dynamisch zur Laufzeit aufgelöst, was starten, stoppen und migrieren von Prozessen zu beliebigen Zeitpunkten erlaubt.

Um unsere Lösung zu Bewerten implementierten wir einen Prototypen auf Basis von InfiniBand. Dieser Prototyp nutzt die Vorteile von InfiniBand, um effiziente Datenübertragungen zu ermöglichen, und vermeidet gleichzeitig die Nachteile von InfiniBand, indem er die Ergebnisse langwieriger Operationen speichert und wiederverwendet. Wir führten Experimente auf Basis dieses Prototypen und des Webservers Jetty durch. Zu diesem Zweck integrierten wir Jetty in das Hadoop map/reduce framework, um realistische Lastbedingungen zu erzeugen. Während dabei die effiziente Integration von LibRIPC und Jetty vergleichsweise einfach war, erwies sich die Integration von LibRIPC und Hadoop als deutlich schwieriger: Um unnötiges Kopieren von Daten zu vermeiden, währen weitgehende Änderungen an der Codebasis von Hadoop erforderlich. Dennoch legen unsere Ergebnisse nahe, dass LibRIPC Datendurchsatz, Latenz und Overhead gegenüber Socketbasierter Kommunikation deutlich verbessert.

Contents

Abstract v

Deutsche Zusammenfassung vii

Contents 1

1 Introduction 3

2 Related Work 7
- 2.1 Socket Improvements 8
- 2.2 Message-based Approaches 9
- 2.3 Local IPC Mechanisms 12
- 2.4 Summary 13

3 LibRIPC Design 15
- 3.1 Analysis 15
  - 3.1.1 Current Cloud Platforms 15
  - 3.1.2 Typical Workloads 17
  - 3.1.3 Traffic Classes 18
- 3.2 Design Goals 18
- 3.3 Basic Architecture 20
- 3.4 Library Interface 21
  - 3.4.1 Short Messages 22
  - 3.4.2 Long Messages 23
  - 3.4.3 Asynchronous Notifications 27
- 3.5 Protocol Mechanics 29
  - 3.5.1 Addressing 29
  - 3.5.2 Buffer Management 30
  - 3.5.3 Waiting for Events 32
  - 3.5.4 Trust 34

4 Application of LibRIPC to a Cloud Web Server 39
4.1 Basic Architecture . . . 39
4.2 Integration of LibRIPC into Jetty . . . 40
4.3 Buffer Handling . . . 41
4.4 Cached Buffers . . . 42
4.5 Summary . . . 42

5 Evaluation 45
5.1 Test System . . . 45
5.2 Test Goals . . . 45
5.3 Prototype Implementation on InfiniBand Hardware . . . 46
5.3.1 Introduction to InfiniBand . . . 46
5.3.2 LibRIPC Service Contexts . . . 49
5.3.3 Short Messages . . . 51
5.3.4 Long Messages . . . 51
5.3.5 Remote DMA Connection Establishment . . . 52
5.3.6 The Resolver . . . 54
5.3.7 Buffer Management . . . 56
5.4 Experimental Results . . . 57
5.4.1 Performance . . . 57
5.4.2 Flexibility . . . 63
5.4.3 Ease of Integration . . . 64
5.4.4 Summary . . . 66

6 Conclusion 67
6.1 Future Work . . . 68

Bibliography 69

Chapter 1

Introduction

Over the last decade, the trend towards cloud computing has been growing throughout the Internet. Customers increasingly tend to host their computing services in the cloud instead of buying and maintaining their own hardware in order to save costs [31, 56]. As a result, large data centers dedicated to hosting these cloud services have emerged. To date, these data centers typically consist of a large number of tightly interconnected commodity computers [8,67]. Since these commodity computers make use of standard CPUs – like Intel Xeon or AMD Opteron – which are optimized for raw computing power, the energy consumption of these data centers is becoming increasingly problematic [38,59]. As a result, new server architectures comprising larger numbers of "weaker" but more cost- and power-efficient CPU cores – like ARM or Intel Atom chips – are starting to emerge [11]. At the same time, features like transport layer protocol offloading, user-level I/O and remote DMA, which are traditionally found only in high-performance interconnects, are currently gaining momentum for more common network architectures like Ethernet as well [12, 34]. These features allow the network hardware to offset the lower processing power of energy-efficient CPUs by carrying out communication-related tasks independently. We therefore expect such features to become increasingly common in datacenter network hardware in the near future. However, despite being subject to extensive research [4, 35, 60, 65], how to use these features in cloud environments in an efficient, generic and easy-to-integrate fashion remains an open question.

Most current cloud platforms are based on commodity operating systems such as Linux or Microsoft Windows. These operating systems are easy to deploy and maintain on the current cloud platforms' commodity hardware, and they are likely to support the customers' applications, as customers would probably run the same operating systems if they were to buy their own, dedicated hardware. While we believe that these operating systems can be ported to future clouds' CPU architectures, they often lack adequate support for the special features we expect future interconnects so incorporate. For example, the network stacks of current commodity operating systems are typically based on the Berkeley socket interface and TCP/IP, neglecting support for offloading-, user-level I/O- or remote DMA-features that may be present in the network hardware. Especially the Berkeley socket interface is largely incompatible with these special features: Socket I/O is synchronous and stream-based, whereas most offloading mechanisms as well as remote DMA are asynchronous and message-based by nature.

There have been numerous efforts investigating the problem of efficiently supporting feature-rich interconnects in commodity operating systems. However, most of these efforts are still based on the socket interface, merely improving the underlying implementation. The Sockets Direct Protocol [26] and Java Fast Sockets [62] build upon sockets, preserving their stream-based semantics, but using remote DMA operations to improve the efficiency of the data transfer. The Superpipeline protocol [18] of the NewMadeleine framework further improves this approach by overlapping the data transfer with its associated overhead – such as memory registration on InfiniBand – in order to further improve both throughput and latency. While the resulting throughput and latency of these solutions are impressive compared to plain TCP/IP networking, we believe the socket interface prevents them from utilizing the full potential of the interconnects they target, mainly because the socket interface's streaming semantics do not preserve information about message boundaries. As a result, large messages are generally fragmented, which introduces overhead that could be avoided by properly using the hardware's offloading features. In addition, none of the approaches mentioned above make proper use of one-sided communication, thus putting unnecessary strain on the CPU.

There have also been research efforts which do not focus on the socket interface, but instead use message-based interfaces which map more closely to the underlying hardware's semantics. The most prominent example is the Message Passing Interface (MPI) [40], which is commonly used in high-performance computing. However, MPI is highly application-specific: It assumes processes using it to be part of the same application, neglecting flexibility and trust issues commonly found in cloud applications. More hardware-specific approaches like InfiniBand Verbs [33] offer the highest possible performance, but since they expose virtually all details about the underlying hardware, they are difficult to use and the resulting applications are not portable to different hardware architectures. The Common Communication Interface (CCI) [6] and NetSlice [45] provide a hardware-independent, yet low-level interface to high-performance hardware. Both offer application portability and high application performance, but since they still expose a high level of detail, they are almost as hard to work with as a hardware-specific interface like InfiniBand verbs. Generally speaking, the message-based solutions currently available lack the flexibility and ease of use the socket interface offers and are therefore a poor fit for cloud applications.

In this thesis, we introduce LibRIPC, a light-weight communication library which leaves the socket interface behind. It provides high network performance in terms of both throughput and latency, while at the same time achieving the application flexibility and ease of use needed by cloud applications. Instead of using sockets, LibRIPC's interface exposes a common denominator of current high-performance interconnects to the application. The level of detail exposed by our interface is low enough to allow for both easy application development – at a level of difficulty similar to socket programming – and application portability across different network architectures. At the same time, the interface stays close enough to the hardware's native semantics to preserve the performance offered by current high performance network architectures.

LibRIPC is built around basic send and receive operations. It features a short- and a long send operation, which are designed to suit control traffic and bulk data transfer, respectively. The short send operation uses the hardware's basic send/receive semantics in order to minimize the message latency, while the long send operation leverages the hardware's remote DMA features to achieve high throughput and to avoid unnecessary copying of the message payload. Hardware-specific details, like connection establishment or memory registration, are generally handled by the library, either completely transparent to the user (connection establishment) or hidden behind a simple and generic function (memory allocation). The results of expensive operations, such as connection establishment or memory registration, are cached and re-used whenever possible in order to minimize overhead. The library uses a hardware- and location-agnostic addressing scheme, which we dubbed service IDs, to address communication endpoints, and resolves these service IDs to hardware specific addresses dynamically. The library thus allows for adding, removing and migrating of services at any time, providing applications with even greater flexibility than IP networking. At the same time, it minimizes application's waiting times by providing low-latency, high-throughput and low-overhead communication.

In order to evaluate LibRIPC's performance, we implemented a prototype on top of InfiniBand hardware. We found that LibRIPC's concepts map well to InfiniBand's hardware semantics, so we were able to use both offloading- and remote DMA functionality extensively. We also constructed a broadcast-based resolver in order to support hardware- and location-independent addressing. Finally, we evaluated our prototype using Jetty [22], a Java-based web server, which we integrated into the Hadoop map/reduce framework [24] in order to create realistic load conditions. In our experiments, we found that LibRIPC was easy to integrate into Jetty and that it outperformed regular socket communication in terms of both throughput and latency.

The rest of this thesis is organized as follows: We will first present some background on the current state of cloud networking as well as related work in Section 2, followed by a detailed description of our library's design in Section 3. In Section 4, we will examine a possible target application and how support for LibRIPC could be integrated into it. Finally, we will present an evaluation of our design in Section 5, before concluding the thesis in Section 6.

Chapter 2

Related Work

The Internet is the largest and arguably the most important computer network ever built. It spans the entire planet, connecting billions of machines; it is a major factor in today's worldwide economy and it continues to grow rapidly. The default protocols on the Internet have been the Transmission Control Protocol (TCP), the User Datagram Protocol (UDP) and the Internet Protocol (IP) ever since the US department of defense adopted them as the standard for all military networks. The adoption of these protocols further advanced when AT&T put the TCP/IP-stack of the UNIX operating system into the public domain in 1989. That TCP/IP included the Berkley socket API as the default programming interface, which as a consequence was also widely adopted. Microsoft followed in 1992 by defining the Winsock-interface, which closely followed the Berkeley socket API. Today, sockets and TCP/IP are the de-facto standard for connecting to the Internet, and implementations of both are included in every general-purpose operating system. As a result of the widespread adoption and interoperability of sockets and TCP/IP, they have gradually replaced other protocols for most forms of local area networking as well.

To date, most applications run in the cloud are regular programs – like web servers, databases or data processing applications – which have been moved to the cloud from dedicated servers in order to save costs [31, 56]. These servers typically run a general-purpose operating system like Linux or Microsoft Windows. As a consequence, networked applications use their target operating system's socket API. In order to minimize deployment effort, most cloud platforms therefore use the same socket abstractions. While this approach has the clear benefit of not requiring changes to the application when moving into the cloud, it comes with a number of disadvantages with respect to performance. First, sockets induce additional latency by copying all data from user to kernel memory on sending and back from kernel to user memory on receiving. Second, sockets handle data as a stream of bytes, which means that any information about message boundaries is lost upon sending. As a result, messages are often fragmented – for example, large files requested from a web server are sent as a stream of TCP packets instead of a single message. Receivers have to issue multiple read calls to their socket until the entire message has been received.

Socket connections are typically built using TCP/IP, which was originally designed to operate over unreliable, wide-area networks such as the modems used in the early days of the Internet. TCP ensures reliability even over such unreliable links, but the complex techniques it employs to achieve reliability induce significant overhead. In a cloud environment, that overhead is often unnecessary, as physical machines are often physically close to each other, and their interconnect provides acceptably low packet loss rates even without TCP's reliability features. Using UDP instead of TCP alleviates the problem, since UDP lacks TCP's reliability mechanisms. However, UDP also does not support fragmentation at the protocol level, which effectively limits the message size to the interconnects maximum transfer unit (MTU).

2.1 Socket Improvements

There have been attempts to alleviate the shortcomings of TCP on high performance interconnects. The **Sockets Direct Protocol** (SDP) [26] available for InfiniBand hardware uses mechanisms similar to our own in order to reduce overhead. It eliminates TCP's overhead by using hardware features to ensure reliability and uses remote DMA to eliminate copying for large payloads. However, SDP uses the Berkeley socket interface and mimics TCP's stream-based semantics to ensure application compatibility, which do not map well to InfiniBand hardware functionality. SDP does not use information about message boundaries and thus fragments large payloads into 128 kilobyte chunks, each of which is transferred using a separate rDMA request. Small payloads, on the other hand, are copied on both send and receive. Finally, in order not to expose special memory handling to the user, SDP pins payload buffers in memory on each send. As a result, SDP's performance stays far below the theoretical maximum of the InfiniBand hardware used in the experiments [26].

The **NewMadeleine** [7] communication framework includes a superpipeline protocol [18] designed to deliver high throughput on high-performance interconnects. It uses remote DMA operations to copy data between pre-allocated buffers and copies data in and out of those buffers on both ends of the connection. However, its pipelined approach mitigates the usual drawbacks of that approach: Data is transferred in chunks, and chunks are copied into or out of a buffer while the next chunk is in transit through the network. As in-memory copying is usually faster than network data transfer, the buffers can never overflow and the copying overhead is completely hidden behind the network data transfer. The approach supports streaming of data and thus maps well to the Berkeley socket interface. By varying the chunk size dynamically, corner cases like the first and last chunk of a data stream are handled efficiently. In total, the data throughput of this pipelined approach is almost identical to that of a raw remote DMA write operation. Unfortunately, two important drawbacks remain: First, the pipelined approach only works efficiently if the amount of data being transferred is large enough for the pipeline to take effect – messages of only a few bytes cannot be broken down into smaller chunks and thus do not benefit from the pipeline at all. Second, for the pipeline to work efficiently, both sides of the connection need to actively copy data either to or from a hardware buffer, which implies overhead in terms of both CPU time and memory bandwidth.

Java Fast Sockets (JFS) [62] follow an approach similar to that of SDP, also copying data twice for small payloads. However, as Java's default socket implementation deals with data in the form of byte arrays, information about message boundaries – namely the array size – is retained, and JFS uses that information to transfer entire arrays in one remote DMA operation without prior serialization. Unfortunately, it is unclear whether that approach also works with data structures other than arrays of primitive data types.

Dalessandro's and Wyckoff's work on **Memory Management Strategies for Data Serving with RDMA** [15] deals with sending the contents of disk files via remote DMA. The performance of sending file data via remote DMA is normally much lower than sending data that is already present in RAM due to disk read time and possibly other required operations such as memory registration. The paper proposes a pipelined approach which overlaps disk reads and memory registration with send operations. The pipeline is implemented in the kernel in order to avoid copying between user- and kernelspace and to reduce the costs of memory registration. Applications invoke the pipeline transparently through the `sendfile()`-system call. While we are not planning to integrate kernel modifications in our current project, we may integrate some insights gained from this work in a future version of our project.

2.2 Message-based Approaches

The **Message Passing Interface** (MPI) [40] is wide-spread on systems featuring high-performance interconnects like InfiniBand. In contrast to sockets, it uses – as the name implies – a message-based interface. MPI's messages are transferred atomically: A message either arrives at the receiver's location in its entirety – which means polling for subsequent packets is never necessary – or not at all. MPI was originally created for high-performance computing, and thus offers both high throughput and low latency. It also uses special hardware features – like remote DMA – to keep the processing overhead as low as possible. However, MPI lacks the flexibility required by most applications. MPI assigns addresses only when the application is started, which implies that all processes wishing to communicate with each other must start at the same time. Starting a process later on or migrating a process to a different physical host is not possible. Fault-tolerance in MPI is also limited; for example, it is not possible to restart a crashed process, forcing applications to revert to methods like checkpointing and rolling back entire partitions [27]. This lack of flexibility alone makes MPI unfit for cloud environments.

InfiniBand Verbs [33] is a low-level software interface created to provide a vendor-independent interface to InfiniBand hardware. It abstracts only from vendor-specific hardware details, but exposes all core InfiniBand functionality directly to the user. Commands are executed directly in the calling process' context, bypassing the operating system kernel if possible. Calls into the kernel and caching issues with memory-mapped I/O registers are handled transparently when necessary. However, the level of detail of the functionality exposed to the user remains high enough to create highly optimized applications and to achieve impressive performance in terms of both bandwidth and latency. On the downside, programming applications using the verbs interface is complex, since hardware-specific data structures like command queues as well as hardware errors must be dealt with explicitly. Also, applications using the verbs libraries are not portable to network architectures other than InfiniBand. The verbs interface is therefore rarely used directly by application programmers. Instead, it primarily serves as a backend for creating higher-level communication libraries like MPI.

NetSlice [45] is a low-level interface to Ethernet network hardware. It provides applications with access to the send and receive queues of the network hardware, bypassing the entire network stack in the operating system kernel. Since a system call is still required to access a device queue, NetSlice implements batching of requests – that is, writing more than one packet to a queue in one operation – in order to minimize system call overhead. It also strives to increase parallelism in packet processing by assigning each queue to a specific CPU core, which then handles all processing of data coming from that queue. While NetSlice achieves an impressive network throughput and scales to high numbers of both network cards and CPU cores [45], it is also difficult to use in ordinary applications. First, it performs no packet processing whatsoever. Instead, it returns raw Ethernet frames; all packet processing is supposed to be done on user level. Second, it requires extensive configuration to achieve maximum performance. For example, the user is expected to manually select which CPU cores are bound to which network queue in order to maximize cache efficiency. We believe that NetSlice is not suitable – or intended – for direct use by applications. Instead, it could serve as a backend for user-level network stacks, packet processors – such as firewalls – or possibly our library.

The **Common Communication Interface** (CCI) [6] aims to provide a low-level, yet hardware independent abstraction of high-performance hardware. It features simple send- and receive functions based on active messages for control traffic, but also exposes explicit remote DMA functionality to the user. While most hardware details are hidden from the user, the CCI requires the application to handle some rather hardware specific tasks which our library performs transparently – like memory registration or the two-step protocol used by our long messages – thus increasing application complexity. In addition, the CCI requires explicit connection establishment between communicating endpoints, which renders process migration more difficult, since the state of an active connection is difficult to migrate.

Illinois Fast Messages [54] are a messaging scheme which extends the concept of active messages [64] to exploit high performance interconnects. The interface of the fast messaging library is similar to our solution in that it features explicit send and receive operations. On the receiver side, messages are buffered until the receiving thread performs a receive operation. Fast messages also exploit hardware features like remote DMA and hardware reliability when possible. However, despite remote DMA being used for the actual data transfer, all messages are copied between send and receive buffers and application memory on both ends of the connection. Since these buffers are pre-allocated to a fixed size, fast messages also suffer from overflowing buffers when large amounts of data are transferred. As a result, fast messages achieve impressive latency results for short messages, but are unfit for bulk data transfer.

FLIP [37] is the network protocol used in Amoeba [51]. It was designed from the ground up to support an operating system spanning multiple machines. It is similar to our own approach in that it addresses processes instead of hosts and dynamically resolves these endpoints to hardware-specific addresses as needed. It uses a message-centric software interface, but it allows fragmentation of messages, which implies that the receiver must potentially reassemble received messages before they can be processed. FLIP strives to minimize protocol overhead, but it does not target interconnects with special features like offloading or remote DMA. Instead of offloading work to the network hardware, FLIP uses active messages [64]: Each message carries a reference to a handler function on the receiver side, which maps particularly well to remote procedure calls, but can significantly reduce the receiver's processing overhead in other cases as well. Unfortunately, active messages can only be implemented in software, which makes the protocol incompatible to hardware offloading.

2.3 Local IPC Mechanisms

FABLE [58] is a library for efficient inter-process communication. It implements efficient zero-copy data transfer between address spaces and is capable of selecting the most efficient method for data transfer automatically. It also features a naming scheme which is independent of the transfer method actually used, which allows the library to keep the method selection as well as related tasks – such as memory management – transparent to the processes using the library. However, FABLE deals with node-local communication only, using shared memory, pipes or loopback network interfaces as means of communication. It may however be possible to extend FABLE's semantics to remote communication by integrating it with our own work.

The **Multikernel** [9] is an operating system which views a single computer as a distributed system. Its kernel assumes that no data is shared between CPU cores. Processes thus do not communicate using shared memory directly, but instead use a message-passing abstraction. The message-passing implementation currently uses shared memory, but is carefully tailored to the cache-coherence protocol used by the hardware. Short messages thus never actually touch memory, which keeps the message latency low. The current multikernel work does not target multi-node systems and is therefore largely orthogonal to our work. However, we believe that it could be extended with remote IPC functionality like our own to form an operating system spanning more than one machine. The authors address that topic, citing the exploitation of "reliable in-order message delivery to substantially simplify its communication" as one of the main problems when moving to distributed Hardware.

K42 [3,39] is a research operating system primarily developed at IBM's Thomas J. Watson Research Center. It is based on a set of server processes running on top of a microkernel. These servers communicate using a set of IPC functions provided by the kernel, including an RPC mechanism dubbed *protected procedure call*, which provided some inspiration for our asynchronous notifications. K42 provides full API and ABI compatibility to Linux and is able to run even large applications like IBM DB2 without modification. While K42 supports NUMA architectures efficiently, it is still intended to run on top of a single machine. However, if the IPC mechanism were extended to cross-node communication, we believe that its various server programs could be distributed across several servers, effectively forming a single operating system spanning multiple machines.

2.4 Summary

The research efforts outlined above can be divided in three categories. The efforts of the first category are built around the Berkeley socket interface. They modify the backend below the socket interface to make better use of the underlying hardware and thus improve network performance. While these approaches are easy to use in applications, their performance generally stays far below the theoretical maximum of the underlying interconnect.

Approaches which do not rely on the socket interface form the second category. These approaches usually offer more complex interfaces, allowing applications to use the underlying hardware more efficiently. Applications using a solution from the second category often outperform applications built around sockets by far. However, the solutions from this category often lack the ease of use and flexibility of sockets, as they are either application-specific – like MPI – or expose a high level of detail, which is not needed by most applications.

The third category comprises local communication solutions, such as IPC mechanisms found in operating systems. While these solutions are largely orthogonal to our work, they provided interesting insights about the functionality needed in LibRIPC. We also view the efforts from the third category as possible applications for our work – using LibRIPC, it may be possible to extend some solutions from the third category to form distributed operating systems.

Chapter 3

LibRIPC Design

In this section, we will present the design of our library and the communication protocol it uses. The description in this section is mostly independent of the underlying interconnect. We assume that remote DMA is supported, but we make no further assumptions about the hardware. Hardware-specific implementation details are discussed in Section 5.

We present an analysis of the type of cloud platform we envision as a target for our library in Section 3.1 and derive the requirements our library needs to fulfill from that analysis in Section 3.2. We then present the design decisions we made to fulfill these requirements: We will start with a quick overview in Section 3.3, followed by a description of the library's interface in Section 3.4 and finally a description of more low-level design choices in Section 3.5.

3.1 Analysis

In this section, we analyze cloud platforms we envision as targets for our library, as well as typical applications running on these platforms. We will start by describing the platforms themselves in Section 3.1.1. We continue with a description of typical applications in Section 3.1.2. We finally present our observations of these application's network usage patterns in Section 3.1.3.

3.1.1 Current Cloud Platforms

Current cloud platforms typically consist of a large number of commodity computers connected by a conventional Ethernet [8, 67]. These computers typically run a wide variety of applications, from web servers over (distributed) databases to peer-to-peer- and high performance computing applications – research efforts on the latter are currently underway [30] –, each of which exhibits its own commu-

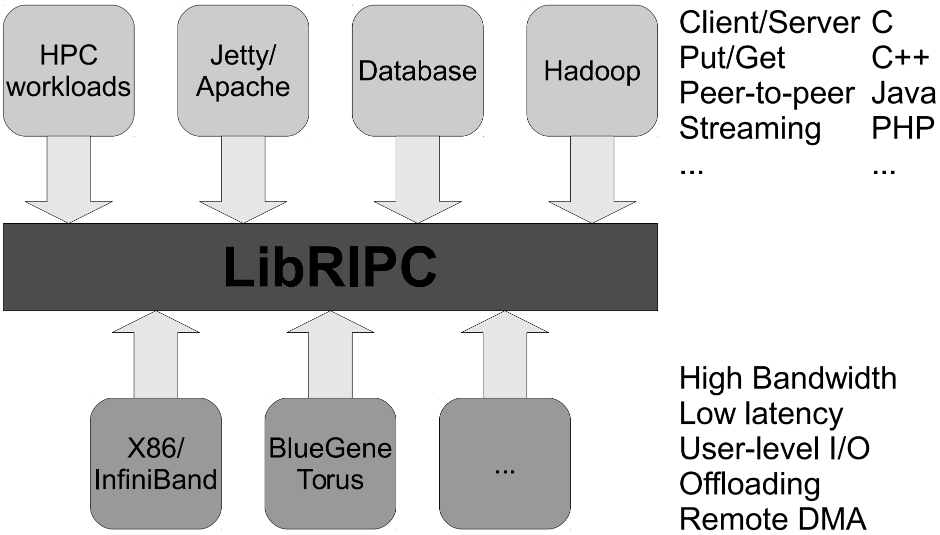


Figure 3.1: LibRIPC bridges the gap between network hardware and applications. It handles the hardware – and its special features – as well as different kinds of applications and programming languages efficiently.

nication patterns, resulting in heterogeneous network traffic throughout the cloud platform. We anticipate that, in the near future, special features formerly reserved for high performance computing systems will become increasingly common in the network interconnects of cloud platforms. These features include direct access to the network devices from user level without operating system involvement, offloading capabilities (the ability of the network hardware to carry out certain tasks independently, without CPU involvement) or remote DMA (the ability to access the main memory of another machine without involving that machine's CPU). Such interconnects can provide both high network performance and application flexibility – however, in order to achieve maximum performance, applications need to be tailored to the specific interconnect they are supposed to use. Low-level interfaces like InfiniBand verbs are available for that purpose. However, these interfaces are difficult to work with, since they expose virtually all hardware details to the programmer. The applications using these interfaces are also not portable to different network architectures.

In order to facilitate application development, it is desirable to have a generic middleware layer which abstracts from the hardware details, as illustrated in Figure 3.1. To date, most applications use sockets and TCP/IP for that purpose. However, the socket API adds a high message latency by copying the data between user and kernel space. TCP adds additional protocol overhead in order to ensure reliability. Communication primitives originating from high performance computing – such as MPI – provide both high bandwidth and low latency, but lack the flexibility required to support peer-to-peer applications. For example, MPI assigns ranks to processes statically at startup time, making it impossible for more processes to join the network at a later time.

3.1.2 Typical Workloads

Cloud platforms are characterized by a wide variety of applications [31], each with its own requirements on the platform's interconnecting network. In order to describe these requirements, we analyze two classes of applications we believe to be representative: Client-server systems and distributed applications.

Client-server-systems are characterized by few servers – frequently, there is only one – serving requests from many clients. A common example on the Internet is a web server, which holds web content in a central location and sends it to clients upon request. As web content can be quite large, such servers often have high bandwidth requirements – ideally, the server needs a network interconnect fast enough to saturate the network interconnects of all clients simultaneously. It is therefore desirable to keep the network's protocol overhead – like packet headers – as low as possible in order to make better use of the server's available bandwidth. Servers often also place high load on their CPU, as the requested data is not always statically available, but needs to be generated on the fly. The transmission of data should therefore not induce a significant CPU overhead, as CPU time is better spent processing requests.

Distributed applications, on the other hand, consist of more than one single server. In a distributed application, the load is typically spread evenly across several hosts, placing lighter load on the individual host than on the server in a client-server-scenario. Distributed databases like MongoDB [1] are a common example of this architecture: Both the data and the processing workload are spread over multiple nodes, with one node acting as a load balancer. On the downside, there is no central location that can maintain all state the system may require in one place. Instead, the individual nodes typically maintain their own state locally and share pieces of it with other nodes as needed. Distributed applications therefore often require frequent synchronization of state between individual hosts. These synchronization messages are often small, but due to their frequency, they can induce significant overhead to the overall system if hosts frequently need to wait for a synchronization message. It is therefore important to keep the message latency as low as possible in order to minimize the waiting overhead. Many distributed applications also require a high degree of flexibility and fault-tolerance, since hosts can join or leave the network or even crash randomly.

3.1.3 Traffic Classes

The two application classes described above demonstrate the need for us to support two distinct types of network traffic, namely control- and data traffic. Control traffic – such as synchronization messages or work requests – typically consists of comparatively small (as low as zero bytes for synchronization messages) but often time-sensitive messages. Due to their small size, control messages do not require high bandwidth. However, since other processes may wait for them, their latency can have a severe impact on the overall system. Data traffic, on the other hand, is characterized by larger amounts of data being transferred at once, which implies that the actual transfer time dominates the latency. Therefore, data traffic benefits more from high network throughput than from reduced latency. A common trait of both types of traffic is that they can both be described as messages. For control messages carrying state, the entire updated state must be received and applied – application of only a part of it would lead to an inconsistent state in the receiver. Data traffic often consists of large data items, each of which is the result of some processing task. For example, the result of a request to a web server is the content of a single (HTML-)file, which can be seen as a message.

3.2 Design Goals

From the traffic patterns described above, we derive a set of goals our library needs to meet in order to efficiently support high performance interconnects in cloud applications. The goals of our library's design are as follows:

Message-based Many protocols exchange messages instead of data streams. We therefore propose to implement an interface based on message passing. The user passes complete messages to the library, which the library then transfers atomically. Messages may be lost or reordered during their transfer, but they are never altered. When a message arrives, it is thus guaranteed to be complete, which makes further polling – as is often necessary with sockets – unnecessary.

Low latency In order to efficiently support control traffic, the library should support a transport optimized for small but latency-sensitive messages. That transport, which we hence name *short send*, uses simple send/receive hardware semantics to transfer data. When using short send, receive buffers are managed by the network hardware, which eliminates the need for initial handshaking between the partners prior to the actual data transfer. The library also avoids copying the data as much as possible in order to further reduce message latency. On the downside, the message size supported by this transport is limited by the hardware receive buffer and/or the network's maximum transfer unit (MTU). We describe the low latency transport in more detail in Section 3.4.1.

High bandwidth The library also offers a second transport optimized for high throughput at the cost of slightly higher latency. This second transport uses the hardware's remote DMA functionality to transfer messages of arbitrary size. It delivers high throughput by eliminating protocol overhead and unnecessary copying of data through the use of remote DMA. On the downside, the remote DMA data transfer requires an initial handshake for the communication partners to agree on the source and destination address of the data transfer, which adds an additional, constant overhead to each message. We describe the high bandwidth transport in more detail in Section 3.4.2.

Low overhead Communication should generally not disturb computation in progress on the same machine. We therefore minimized the overhead of our library wherever possible. The library generally avoids copying of data as much as possible and caches hardware resources for later re-use to avoid expensive allocation operations. Since remote DMA operations are transparent to one side of the communication, they can also reduce overhead if applied carefully. For example, a server under heavy load does not need to spend CPU time on sending data to its clients. Instead, it performs the initial handshake for a remote DMA transfer, and then processes the next request, while the client handles the data transfer transparently to the server process.

Portability Our library provides an interface that is completely independent of the underlying hardware. Applications using our library are therefore easily ported to a different platform, as it is not necessary to make modifications to the application in order to support a new network interconnect. However, the library was designed not to sacrifice performance in order to achieve hardware independence. Instead, its interface exposes features various hardware interconnects have in common to the user, while handling hardware-specific details transparently to the application.

Ease of integration Our library is easy to integrate into existing applications. It exposes a fairly small set of primitives, allowing programmers to quickly learn how to use it. In addition, the library can be used from various different programming languages, including managed runtimes such as Java.

Flexibility Applications using our library fully support the flexibility of cloud environments. The library supports starting, stopping, migrating and crashing

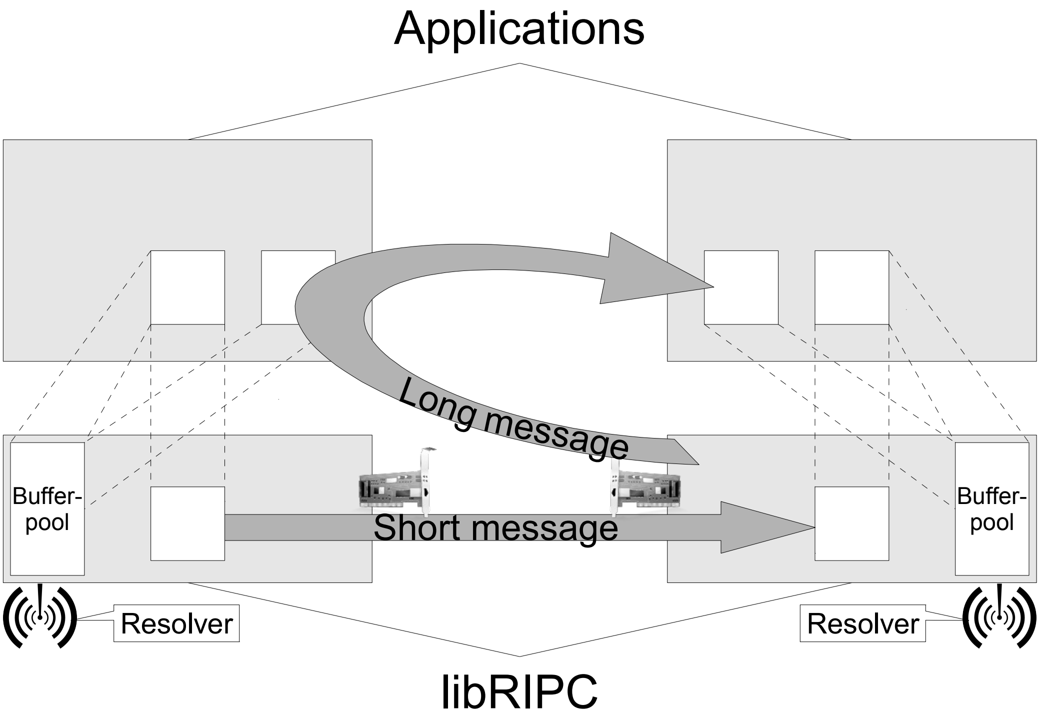


Figure 3.2: Overview of LibRIPC

of application instances at any time. Runtime networking information, such as hardware addresses, is resolved dynamically as needed.

3.3 Basic Architecture

Our library is based on messages rather than streams of bytes, and it transfers messages atomically. This implies potentially lower waiting times at the receiving end – messages are always transferred in their entirety, so there is no need to wait for more data on a socket. Our interface offers two send functions for short and long messages, respectively. The long send function was designed to accommodate data traffic and thus can transfer messages of any size. It delivers high throughput by eliminating protocol overhead and unnecessary copying of data through the use of remote DMA. Since remote DMA is also transparent to one side of the connection, long sends also reduce the CPU load of one of the communication partners – in a client-server scenario, this reduced CPU overhead permits the server to handle more requests per time. The short send function accommodates control traffic by using simple send/receive hardware semantics instead of remote DMA. It also eliminates as much copying as the hardware permits. It delivers a lower latency than the long send function since it does not suffer the additional startup latency of remote DMA, though the maximum size of messages is limited as a result. The standard receive function of our library blocks until a message arrives, which allows servers to wait for incoming requests without wasting CPU cycles. However, since relying on the cooperation of the receiving thread is not always desirable, our library also features an asynchronous messaging facility. When an asynchronous message is received, the control flow of the receiving process is interrupted and a previously registered callback function is called. This mechanism allows the user to efficiently support asynchronous events, which would otherwise require blocking for extended periods of time.

The library strives to minimize the overhead associated with the data transfer; especially, it avoids copying the transmitted data as much as possible. Data is always passed to the library as pointers; data is then read directly from its location in the user's address space. Conversely, the receive function places received data directly in the user's address space and returns a pointer when the transfer is finished. Though not all network architectures allow us to avoid copying entirely, our library makes extensive use of special features of the network hardware – such as remote DMA – to reduce overhead.

While the library makes extensive use of special features of the network hardware, it hides its use of these features from the user, providing an interface that is completely hardware-independent. The hardware-independence allows the user to benefit from the full performance of the available network interconnect, while not depending on any specific network architecture. Specifically, when the network architecture changes, only the library needs to be re-written, while user software using it continues to operate without modification. Furthermore, the abstraction from hardware details makes the interface straightforward to use, which simplifies the integration of our library with existing software.

The library assigns addresses services (i.e. programs) rather than machines, allowing services to join or leave the network or to migrate from one machine to another, without disturbing communication. The addresses, which we call *service IDs*, are dynamically resolved to hardware-specific addresses as necessary.

3.4 Library Interface

In this section, we will present the user interface of our library and the rationale behind its design in more detail. We will describe our short message protocol in Section 3.4.1, the long message protocol in Section 3.4.2 and the asynchronous notification messages in Section 3.4.3. Following this section, we will discuss lower-level design issues that arise from implementing the interface.

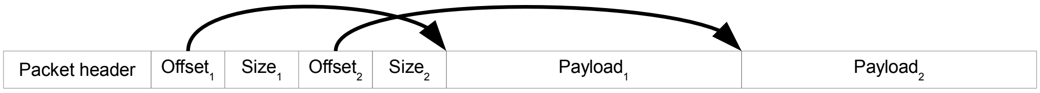


Figure 3.3: Layout of a short message packet. The header fields in front of the payload are fixed-size and can therefore be processed efficiently.

3.4.1 Short Messages

The short message protocol was designed mainly for control messages, such as work requests or synchronization messages. Control messages are usually small, but tend to have tight latency requirements. They are also often unsolicited, which implies the need for fault-tolerance, as the destination service of a control message might have terminated or migrated since the client last contacted it.

Short messages are well accommodated by the direct send functionality of the network hardware, which is supported by every network architecture we investigated. When using direct send, the network hardware at the receiver side places the data in a pre-defined receive buffer, ensuring that no previously received data is accidentally overwritten. The main advantage of this protocol over the long send protocol is that it requires no negotiation between the sender and receiver before data is transferred, which lowers the overall latency. However, the amount of data per message is limited by the receive buffer size and possibly other factors, such as the network's Maximum Transfer Unit (MTU).

The layout of a short message is depicted in Figure 3.3. In order to create a short message, the client needs to concatenate all data items contained in the message into a single buffer. This can either be done manually by copying all data items into the send buffer or the hardware's scatter-gather functionality may be used if present. The buffer is then prepended with a header describing the message's contents. The header consists of a static part, containing the source and destination service IDs of the message and the number of data items in the message, and an array containing the sizes and offsets of the individual payload items, calculated from the beginning of the header. From that information, the receiver can quickly extract the original data items. The message size, including the header, is limited by the receive buffer size and/or the MTU. If a message is larger than that threshold, the data transfer will fail as messages are not automatically fragmented.

In order to achieve low latency, the implementation of the short send protocol should avoid copying data whenever possible. Fortunately, it is common for high performance network architectures to allow user-level access to send and receive buffers. If such access is possible, it can be used to send the message payload directly from the sender's to the receiver's address space. Payload buffers are passed to and from our library as pointers only; the network hardware then reads the payload directly from the sender's address space and stores received data in buffers mapped into the receiver's address space, completely eliminating the copying overhead. On architectures where this is not possible, we have no choice but to revert to copying – however, since the size of short messages are limited, we believe the overhead induced by copying to be tolerable.

To further lower the latency, we also decided not to implement reliability for short sends, but to drop messages silently on transmission failures. Reliability requires some form of acknowledgment on packet reception, which can add considerably to message latency. In addition, reliability is not needed for all applications. We therefore leave that choice to the users of our library by allowing them to implement their own, reliable protocol on top of our library if needed.

3.4.2 Long Messages

We designed the long message protocol for data messages carrying payloads larger than the receive buffer size and/or the network MTU. Guaranteeing high throughput, reasonably low latency and atomicity[1] all at the same time requires a different approach than for short messages. In order to achieve atomicity, we need to ensure that, if the message is fragmented, no individual packets are lost or reordered during transmission. We therefore need to use a reliable transport if one is offered by the hardware, or implement one ourselves. The latency of long messages is dominated by the time needed to transfer their payload, which is potentially large. We therefore need to avoid adding protocol metadata – such as packet headers – during the transfer, as any increase in the total amount of data will also increase the total transfer time. On the other hand, an additional, constant setup overhead may be tolerable if it is small compared to the transfer time.

Our long message protocol uses the remote DMA functionality offered by the network hardware. As a first step, a client wishing to send a long message puts the message's contents into a buffer accessible by the network hardware. Depending on the type of hardware used, this may involve registering a buffer with the network hardware or allocating a buffer with special properties – such as using physically contiguous memory. The client then sends a control message to the intended recipient via the same send/receive semantics that are also used for short messages. That message contains the same header as a short message, but carries a data structure describing one or more data buffers on the sender side instead of the message content itself. Once that descriptor item has been received, the recipient performs a remote DMA read operation for each descriptor item to copy the data from the sender's memory directly into a separate buffer called

[1] By atomicity, we mean that messages are either received in their entirety or not at all.

a *receive window* – a concept borrowed from [43] – in its own address space. The user is free to specify the location and size of the receive windows, which implies that the size of long messages is not limited by the hardware buffer used for receiving short- and control messages.

An alternative to having the receiver pull the data from the sender's memory is to have the sender push the data into the receiver's memory using a remote DMA write operation. However, in order to use write instead of read operations, we would need a pre-allocated receive window known to each sender before the send operation begins. Unfortunately, it is not possible to share such a receive window between multiple senders [55]: If multiple clients were to send data into the same receive window, they would not know about each other's send operations and therefore likely overwrite each other's data. Using non-shared receive windows instead would require one receive window for each possible sender, which would cause an immense memory footprint as the number of nodes in the network grows large. In addition, the sizes of incoming messages are not known beforehand, so any statically-allocated receive window would imply an upper bound to message size. We therefore decided to use an additional control message for negotiation and then allocate memory dynamically. That approach allows for both security – there is no sharing of buffers – and unlimited message size, while at the same time keeping the memory footprint reasonable by allocating memory only as needed.

Prior to receiving long messages, the user needs to declare arbitrary regions of memory as receive windows. The payload of received long messages is then placed in these receive windows – each long message can contain several data items, each of which is placed in its own window. Data items for which no receive window is available are discarded. There are two reasons for this requirement. The first reason is security: If no receive windows were used and the library would instead just allocate the memory necessary for each message, an attacker could easily consume all available RAM by sending many large messages. Receive windows allow the receiver to control how much memory he is willing to use for long messages[2]. The second reason is the ability to control the exact location where the received data is placed. If, for example, a user wishes to receive only a portion of a large array, he can declare that portion as a receive window, causing the received data to be placed in the right location without the need for copying. If the size of the data is known to the receiver, it is even possible to achieve a scatter-gather-like data transfer by posting multiple, adjacent receive windows of the same size as the data items. The receive window placement will then cause the data items to be stored adjacent to each other, regardless of their original location in the sender's memory. This flexibility of data placement can hold a significant performance advantage if applied carefully.

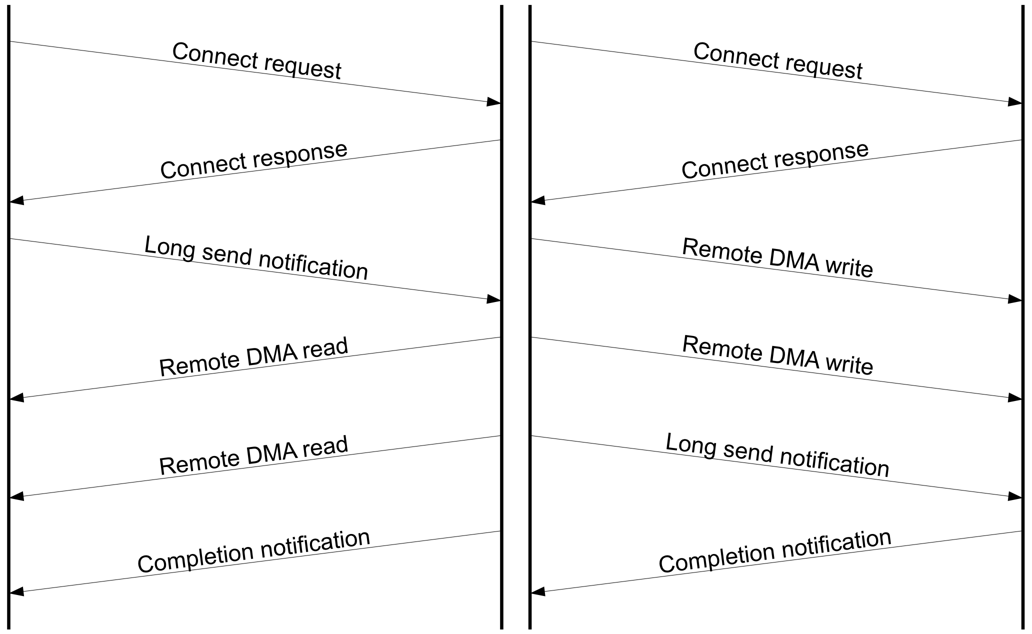
[2] See Section 3.5.4 for more details

The entire process of sending a long message containing two data items is depicted in Figure 3.4(a). Note that the first two packets are necessary on most – but not all – network architectures in order to create a reliable transport between the two endpoints. However, this connection establishment is only necessary on the first long message between the two endpoints, as the connection can stay active after the first message so it can be re-used for subsequent messages. The completion notification at the end of the communication process is necessary since the sender is not notified about successful completion of remote DMA read operations and thus has no way of knowing when it is safe to re-use the buffer(s) he just sent. Our library does not send that completion message automatically, since we believe the application to be the best place to decide how the completion can be implemented most efficiently. For example, the completion notification need not be a dedicated message. Instead, the user could attach the completion message to other messages of his application protocol – something our library is unable to do as it does not understand the application's protocol. Depending on the application, it might also be viable to use an entirely different mechanism, such as timeouts, in place of completion messages.

Compared to short sends, this protocol requires at least one additional message, which incurs additional latency. However, this latency is mitigated by the speed and size independence of the data transfer. In terms of speed, remote DMA data transfers are not easy to beat: Once a remote DMA data transfer is set up, the data is transferred at raw speed; no software packet headers are necessary, and all data processing – such as fragmentation – is done by the network hardware. In addition, using remote DMA, the data is always transferred directly from the sender's to the receiver's address space, with no need for additional copying. We believe that performance advantage outweighs the additional setup message as long as the buffer to be transferred is large enough so the actual data transfer time dominates the total latency. Remote DMA data transfers are also independent of the size of the data being transferred: Any amount of data that fits into the physical RAM can be read or written, which means that our long send protocol will work with messages of any size.

Return Buffers

When using the regular long send protocol described in Section 3.4.2, all work associated with the actual data transfer – namely the setup of the remote DMA read operation – is done by the receiver of the message, while the data transfer is completely transparent to the sender. However, depending on the application, it may be desirable to have the sender do that work by using a remote DMA write operation. Unfortunately, using write instead of read operations is potentially unsafe [55] and therefore not easy to implement. Allowing multiple clients to



(a) Without return buffers (b) Using return buffers

Figure 3.4: Messages needed for a long send containing two data items with no pre-existing connection. The initial handshake must take place before any data or notification messages can be sent. The completion notification tells the sender that it is safe to re-use the buffers used for send the message.

write into the same buffer would cause these clients to overwrite each other's data since they do not know about each other's writes. Using a dedicated buffer for each client solves that problem. However, pre-allocating a buffer for each client that might send a long message would result in a large memory footprint. It is therefore desirable to allocate buffers only for those clients that actually need them.

In order to enable remote DMA writes in a safe way while at the same time keeping the memory footprint reasonable, we came up with the concept of *return buffers*. The concept of return buffers is similar to the one used in [53]. A client expecting a reply to his messages can allocate a number of return buffers and then attach descriptor items describing these return buffers to its messages. The recipient of the message remembers the information about these return buffers until it sends a reply. When it later sends a reply via long message it issues a remote DMA write request into one or more of the return buffers, followed by a notification message. The procedure of sending a long message containing two data items using return buffers is depicted in Figure 3.4(b). Note that the completion message is not strictly necessary in this case, as buffers that have been written directly into return buffers are immediately safe to re-use. However, return buffers are handled by our library, which means that the sender is currently not aware of return buffers the receiver may have posted; the sender therefore has no choice but to wait for a completion notification before re-using the send buffers in order to ensure that no data is accidentally overwritten before the send operation is complete.

Return buffers by themselves do not reduce the latency of long messages, as the same number of round trips is required with and without return buffers. However, they allow the user to use the fact that remote DMA data transfers are transparent to the communication partner to his advantage – or, in other words, to control which end of the connection does the work necessary to transfer the data. For example, suppose a client, under heavy load, sends a request to a lightly loaded server and then resumes other work while waiting for a reply. Using return buffers, the server writes the reply into the client's memory without disturbing the computation in the client. When the client finally calls the receive function, the data transfer is already complete and computation can resume immediately. Conversely, a heavily loaded server may choose to ignore the return buffers. Instead, it simply places the reply in its own memory, sends a notification message and proceeds to processing the next request, leaving the work necessary for the actual data transfer to the client. Summing up, return buffers are a tool for programmers so achieve better load balancing and thus increase overall throughput.

3.4.3 Asynchronous Notifications

It is not always desirable to block while waiting for a message from a remote service. For example, when a client under heavy load sends a lengthy request to a server, that client could resume computation while the request is being processed. However, the default receive function of our library blocks until a message arrives, which leaves the client with the choice of either suspending computation and waiting for the message or to resume computation and possibly leave the server's reply in the receive buffer, delaying the processing of the server's reply.

In order to solve this problem, our library features an asynchronous messaging facility. Using asynchronous messages, the server from our previous example can interrupt the client when it has finished processing the client's request, and the client can resume computation undisturbed until the asynchronous notification arrives. When the notification arrives, the client issues a receive call knowing the data it waited for is ready. Asynchronous notifications thus ensure that the server's reply is processed upon arrival without additional delay and at the same time eliminates the need for blocking in the client.

The asynchronous messaging facility in our library is based on callbacks. A service wishing to receive asynchronous notifications first registers a callback function with our library. The library issues a handle – similar to a file descriptor

– and returns it to the calling service. The service then passes the handle to any other service he wishes to receive asynchronous notifications from.

When a service wishes to send an asynchronous notification, the library first crafts a special message, which contains the callback handle the library previously returned to the receiving process as well as any arguments the callback function may require. It then sends that message to a special handler thread which runs in the address space of the receiving service and is managed by our library. The library uses the hardware's send/receive semantics to transfer the message – similar as for short messages – in order to minimize its latency. As a side effect, the total size of the message, including the callback function's arguments, is limited to the size of a short message. We believe this restriction to be acceptable, as the user can still transfer larger amounts of data as a long message received by the callback function if necessary. Also note that different mechanisms for transferring the notification message may be possible on some network architectures. On BlueGene [36], for example, the global interrupt network could be used to transfer the notification message, which would provide much lower latency than short messages.

The receiver thread subsequently looks up the appropriate callback function using the handle contained in the notification message and calls it in his own context. We chose to run the callback in the context of the receiver thread because that approach does not incur any additional latency beyond a function call, and it does not require interrupting any computation that may be in progress when the notification message arrives. However, this approach also implies the possibility of the callback and other program code running concurrently, which means that the programmer needs to consider thread-safety issues even if his program is otherwise single-threaded. As an alternative, the callback could be started in the context of the thread that originally registered it. On Linux, such a delegated function call could be implemented using POSIX signals, which can be directed to a specific thread using the function `pthread_sigqueue()` [44]. The callback would ultimately be run as a signal handler in the context of the registering thread, thus interrupting other computation in that thread until the handler returns. While this approach does not imply concurrency, the real-time signal adds additional latency to the message. Since we expect the callback functions to typically be small – possibly a single receive statement when the asynchronous message is used to signal the completion of a lengthy operation – we believe that additional latency outweighs the need for synchronization implied by calling the callback function in the receiver thread.

3.5 Protocol Mechanics

In this section, we will discuss the design decisions we made for the lower-level protocol of our library. In contrast to the library's interface, most mechanisms presented in this section are not directly visible to the user, but they are nonetheless important for the final library's performance. We will first discuss the library's addressing and name resolution scheme in Section 3.5.1. We will then present the library's buffer management in Section 3.5.2 and discuss methods for waiting on external events in Section 3.5.3. Finally, we discuss issues arising from communicating with untrusted applications in Section 3.5.4.

3.5.1 Addressing

Our library needs a flexible addressing scheme in order to allow addition, removal and migration of participating programs at runtime. While addition and removal are relatively straightforward – all that is needed is some form of dynamic address resolution – migration is not always trivial. Most protocols use a pair of machine- and service address to address endpoints – an example of this is the IP address:port pair used on the Internet. However, a process migrating from one machine to another can not take the machine's address with it and must therefore notify its previous communication partners of the address change.

To facilitate migration, we chose to opt for a location-independent form of addressing. Services are addressed by *service IDs*. Those service IDs can be thought of as global process IDs. They need to be unique throughout the network, but are independent of the physical location of the process. Resolution of service IDs to hardware addresses – which is necessary for routing purposes – is done dynamically as needed. When a process migrates, it simply keeps its previous service ID; if a message is sent to it later, that send will fail as the physical endpoint the service ID was previously resolved to no longer exists. In that case, the sender simply needs to resolve the same service ID again to obtain the new physical location. Service addition and removal are also supported naturally by this scheme, with a removal being detected when the additional resolve request after a send failure yields no answer.

Resolving service IDs to hardware addresses can be done in a variety of ways. A straightforward solution would be to use a standard Domain Name Service [49, 50], which can serve arbitrary data through so-called TXT records. However, DNS answers are supposed to be cached – each response has an associated time to live – which would delay the propagation of changes after events like process migrations. Distributed lock managers like Chubby [10] or its open-source equivalent Zookeeper [32] can act as a distributed name service. They offer the required flexibility and scale well, which makes them an ideal fit for cloud environments[3]. However, since naming of services is not the primary goal of this work, we decided that distributed lock managers are too complex for our purposes at the moment – Chubby's client library amounts to approximately 7000 lines of code [10] – though they may be the solution of choice once our library is applied to more complex settings.

Since resolving is not the primary goal of this work but only a means to an end, we chose to keep the resolver simple for the time being. We opted for a broadcast-based resolver similar to the scheme used in Amoeba [51]. Whenever one service wishes to send a message to another service of which it does not yet know the hardware address, it sends a broadcast message to all services on the network. That broadcast message contains the service ID being queried, the service ID of the querying thread, all information necessary to contact the querying service and all information necessary to send the reply via unicast, in order to reduce the number of broadcast messages. The broadcast is received and processed by a separate *responder thread* in each service. The service with the ID being requested then sends a unicast reply containing the necessary hardware address information; all other services simply discard the message. To reduce the total number of broadcast messages, it is also possible to attach the sender's hardware address information to the broadcast request. All services receiving the broadcast can then cache the information in case they want to contact the requesting service later on. Note that this scheme is only one possible solution to the address resolution problem – there exists a plethora of different solutions, like distributed coordination frameworks [10, 32] services or distributed hash tables [61]. Any one of these solutions will work as long as the data is kept up to date on service addition, removal or migration.

3.5.2 Buffer Management

Many network architectures require some form of memory registration or allocation of memory with special properties before that memory can be used by the network hardware. For example, architectures supporting remote DMA often require the user to explicitly share a segment of memory before it can be accessed by remote machines. Other architectures sometimes require memory to be allocated as physically contiguous chunks, so they can use the local DMA engine to access it.

To accommodate these special memory needs, the library needs a memory allocation function of its own, which, depending on the exact needs of the network hardware present, can be either a completely new allocator or an extension to the one already present in the operating system. Contiguous chunks of memory, for

[3] In some of Google's services, Chubby has completely replaced DNS.

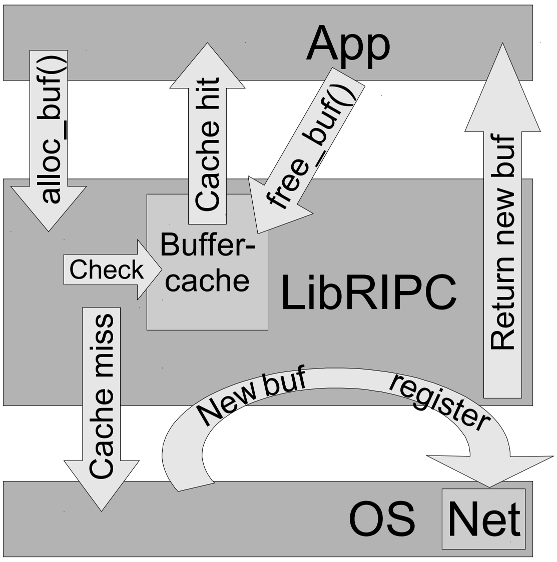


Figure 3.5: Buffer allocation and -caching in LibRIPC. The OS and the network hardware are only involved on cache misses.

example, could be obtained from the bigphysarea driver [47] – a driver which allows users to allocate chunks of physically contiguous memory – via `mmap()`. On architectures requiring memory registration only, memory can simply be obtained from the system's `malloc()` function, registered and returned to the user. In either case, these extra steps need to be hidden from the user in order to preserve the application's independence from the specific network architecture. Our basic idea to achieve this is to provide an allocation function the user is supposed to use whenever he allocates a buffer that might later be sent over the network. These buffers can, however, be used for any other purpose as well, which allows the user to just use our allocation function if in doubt.

It is important to remember that memory registration or allocating memory with special properties can be an expensive operation [48]. Our library therefore needs a way for users to return memory to it, as well as some form of internal buffer management, allowing it to re-use buffers returned by the user instead of allocating new ones. To this end, the library does not immediately free buffers returned to it, but instead remembers them for later re-use. When the user issues a request for a new memory buffer, the library first checks if a suitable buffer is already available. If it finds a buffer big enough to accommodate the user's request, it returns this buffer; otherwise, it allocates a new buffer. The entire process of allocating a buffer is depicted in Figure 3.5.

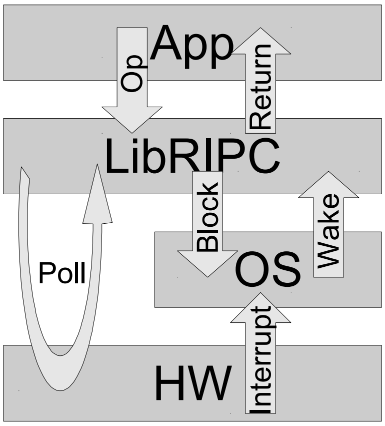


Figure 3.6: Polling vs. interrupt-driven waiting. Polling bypasses the OS, thus achieving much lower latencies as long as the process is not preempted. When blocking the process instead, the hardware interrupt and the reactivation of the blocked process add to the total latency – however, the process can be woken any time, while a preempted polling process does not notice the event it is polling for until it is reactivated by the scheduler.

3.5.3 Waiting for Events

Our library frequently needs to wait for hardware events to occur. These events can be either local – like waiting for the completion of a send operation offloaded to the hardware – or remote – like the arrival of a message from a remote service. Both the length of the waiting periods and the predictability of that length vary widely with the type of event being waited for. For example, the time needed to send a short message is typically in the microsecond range, while waiting for a message from another service can literally take forever if that service has crashed.

Waiting for an event can be implemented in two ways: Either by polling the hardware or by blocking and waiting for an interrupt. Both methods are illustrated in Figure 3.6. Polling in a tight loop can achieve extremely short reaction times – especially if the network device can be accessed directly from user level – but generates high CPU load. In addition, a polling process may be preempted before the event it is polling for occurs. In that case, the process will not be woken when the event occurs. Instead, it has to wait until it is reactivated by the scheduler. Polling is therefore suitable if the waiting period is expected to be short. Blocking, on the other hand, does not generate any CPU load, but adds some extra latency – namely at least that of the interrupt handler – to the reaction time. To make matters worse, other threads that are running when the interrupt occurs might not be preempted until the next timer tick, which can add latency that is orders of magnitude higher than that of the event being waited for[4]. Blocking is therefore suitable for longer waiting times.

Our library currently waits for the completion of short- and control message send operation. This waiting is necessary on some architectures to catch hardware errors – on InfiniBand, for example, queue pairs transition to an error state on send failures and need to be reset before they can be used again. However, as short messages are unreliable, we only need to wait for feedback from the local hardware, which – as messages are small – typically arrives after few microseconds. Furthermore, all network architectures we examined generate an event at the end of a send operation, even if the operation is unsuccessful. The total wait time is therefore guaranteed to be much shorter than the timeslice length, which keeps the CPU load generated by polling for such events acceptably low. We therefore decided to always poll for such short local completion events.

Things are different for long messages. As for short messages, the library needs to wait for the completion of the remote DMA read operation that transfers the actual payload in order to catch possible hardware errors. However, since the payload of a long message can be arbitrarily large, that remote DMA read operation can take a long time to complete. Polling during that time can lead to high CPU load for large messages. However, it is important to remember that long messages need not necessarily be long – it is also possible to send long messages with payloads as small as one byte, in which case the interrupt latency incurred by blocking would unduly increase the message latency. We believe that a dynamic approach is suitable to solve this dilemma: Since the size of the payload is known at the time the remote DMA operation is started, the library can dynamically decide whether to poll or block until the operation completes.

Some control messages – like resolver requests – are a special case, as the library also needs to wait for a reply from a remote service. As control messages are small – they are typically less than 100 bytes long – their round trip time on InfiniBand is only around 20 microseconds. They are also processed quickly in the receiver, provided sufficient CPU resources are available – if the CPU is expected to be busy most of the time, it is desirable to process control messages in a separate thread with a high scheduling priority. Replies to control messages are therefore expected to arrive quickly, which warrants polling for reply messages. However, like short messages, control messages are unreliable. If either the request or the reply is lost in transit, the requesting service will spin until the request times out, causing high CPU load in the meantime. However, since the physical nodes are relatively close to each other, we expect the packet loss rates to be reasonably low. In addition, since replies are expected to arrive within a short timeframe, timeouts can safely expire after relatively short periods, which limits the amount

[4] The timer tick frequency of the Linux kernel is typically between 1 and 10 ms.

of wasted CPU cycles per lost message. There are also network architectures – like BlueGene's torus – which offer reliability as a hardware feature and thus do not suffer from this problem at all. We therefore decided to poll for control message replies as well.

However, polling for messages from a remote party is only viable if the message being waited for is expected to arrive within a short timeframe. This expectation is reasonable for control messages, but not for the general case of one service waiting for a message from another. For example, a client sending a request to a web server has no way of knowing how long the processing of that request is going to take – or if it is being processed at all, as the server may have crashed. A thread calling the receive function of our library is therefore always blocked until a message arrives. However, the receive function can be interrupted – for example by a signal – which allows the user to implement his own mechanism for handling crashed servers.

3.5.4 Trust

Cloud environments are often characterized by many different users running their applications on the same platform. As both the applications and the users owning them are not necessarily known, it would be dangerous to place ultimate trust in remote applications when communicating with them. Our library therefore contains mechanisms which allow the application to communicate safely with untrusted parties. In this section, we highlight these mechanisms, as well as some vulnerabilities which are left in our design, but are not easily fixable.

Resource Exhaustion

A simple, yet effective way of disabling a machine on a network is a denial of service attack. In its simplest form, it only involves overwhelming the target with more network packets than it can handle. If the attacker has enough network bandwidth at his disposal, there is no effective defense against this type of attack. Massively complex firewalls and intrusion detection systems are in place to protect companies and government agencies from this type of attack from the Internet, yet their websites are frequently taken down by hackers [41].

Similar attacks are possible against processes using our library. As the size of the receive buffer for short- and control messages is usually limited, it is possible to consume the entire buffer by sending messages to the target with a high frequency. As soon as the entire buffer is exhausted, the target can no longer receive messages until it retrieves the attacker's messages from the buffer, as messages for which no buffer space is available are simply dropped. Similarly, it is possible to consume all posted receive windows for long messages, causing subsequent long messages to be rejected.

There are mitigation strategies against resource exhaustion attacks [42]. For example, it is desirable to have a separate receive buffer for each service. That way, exhaustion of a receive buffer only prevents the service owning the buffer from receiving messages, while other services running on the same machine remain unaffected. However, if the total number of receive buffers is limited, it may be necessary to share buffers between multiple services. In that case, all services using the buffer being attacked affected by the attack. If there is no limitation on the total number of buffers, it is also possible to have each service allocate separate receive buffers for each sender. That way, an attacker could also exhaust receive buffers dedicated to the attacker specifically, while communication of all other services would remain undisturbed. However, that approach would likely not scale well, as the amount of memory required would grow quadratically with the number of services on the network – for n services, each service would need n-1 receive buffers.

Some software displays even worse vulnerabilities to denial of service attacks. On some targets, an attacker can cause a server program to consume all available RAM or CPU time, thus affecting other processes as well or even disabling the machine altogether. The limited receive buffer size of our library defeats this type of attack effectively – an attacker can never consume more memory than is allocated as receive buffers. Furthermore, if the receive buffer is exhausted, additional network packets are dropped by the network card and thus do not cause additional processing overhead. Thus, additional computation on a machine being attacked should remain mostly undisturbed. The use of receive windows ensures the same for long messages. If the library would allocate memory for incoming long messages automatically, an attacker could easily consume all available RAM just by sending enough messages. Through receive windows, the user can instead control how much memory he is willing to devote to incoming messages.

Cross-client Information Disclosure

Our library makes extensive use of remote DMA in order to increase performance. While the performance advantage is substantial, remote DMA can potentially pose a security risk, since memory regions shared through remote DMA are usually accessible not only to the intended communication partner, but to other machines as well. This may lead to stealing of information from another process' buffer or even manipulation of that information if the buffer is shared for writing.

It is, however, important to note that information about shared buffers is not usually disclosed to any process other than the intended communication partner[5]. The attacker therefore needs to guess the relevant information about the buffer he is attempting to access. The exact nature of that information varies between different network architectures. For example, InfiniBand requires the attacker to guess the buffer's address in the victim's address space as well as a randomly assigned remote access key, while on BlueGene's torus network, guessing only a counter and fifo number – both of which are comparatively small [36] – is sufficient. Fortunately, all architectures we examined generate an interrupt on the machine owning the buffer if the attacker's guess is incorrect, which facilitates detection of this type of attack as the attacker is unlikely to succeed on the first try. In addition, numbers that can be guessed are randomized by our library in order to increase the chance of detecting malicious accesses. However, if the space from which information needs to be guessed is sufficiently small, an attacker could consume a large portion of that space – for example most counters on a BlueGene system – thus making sure that most random accesses are valid. Since valid remote DMA requests go unnoticed by the victim, he could then try to access confidential information through trial and error.

We are not aware of a viable solution for architectures which do not have appropriate security features built into their hardware. It is therefore important for users of our library to be aware of this potential threat when designing their applications.

Resolver Cache Poisoning

Our current, broadcast-based resolver caches all information it receives about remote processes in order to reduce the number of broadcast messages on the network. Unfortunately, this automatic caching can be exploited to re-route messages, allowing for man-in-the-middle-attacks. The attack scheme is similar to ARP spoofing [63]: The attacker sends a resolver request, containing the attacker's hardware address and the target service ID as source information. Any instance of our library receiving this request will then update its cache, causing all traffic intended for the target to be routed to the attacker. Alternatively, if services the broadcast is not intended for would not update their caches, the attacker could wait for other processes to resolve the target and reply with his own information – the malicious reply taking precedence if it is received by the requester before the real one. Figure 3.7 illustrates the principle of the attack.

To date, there is no effective protection from ARP spoofing attacks, though numerous attempts have been made to solve the problem [2]. The root of the problem is that neither requests nor replies are authenticated, which implies that

[5] Provided the underlying network routes packets only between the two endpoints. If packets are broadcast to all machines in the network, buffer information is easily intercepted.

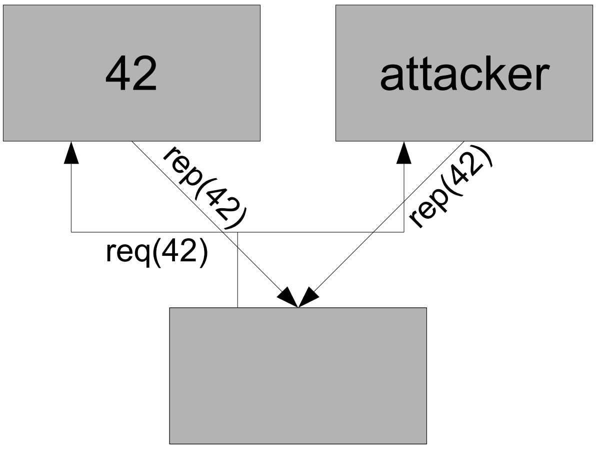


Figure 3.7: Resolver cache poisoning. If the attacker's reply arrives before the real one, the attack can impersonate service ID 42.

all parties must blindly trust any message they receive. The same is the case for our current resolver, so we believe the problem to be equally hard to solve. In order to trust incoming resolver replies, we would either need to authenticate each reply or – if available – to transmit the replies over a different channel that is inherently trusted.

Fortunately, the current resolver was designed from the ground up to be easily replaceable, the main reasons for its current design being its flexibility and relative ease of implementation. An alternative would be a central directory server knowing the service ID to physical address mapping of each service on the network, with the resolve function in our library being merely a client to that server, similar to the Domain Name System (DNS) [49,50]. Using a central service would make manipulations much harder, since i) clients would no longer have to trust every resolver reply, since trustworthy replies are always sent from the same location and ii) the server would not have to trust every update message – for example, notifications about process migrations could be accepted only if they originate from the management software that initiated the migration – which would make it much harder for an attacker to inject information. A solution based on a central server could also incorporate security extensions similar to those proposed for DNS [5,17].

Chapter 4

Application of LibRIPC to a Cloud Web Server

In this section, we will present some applications which we believe will benefit from using our library. We will also give some details as to how support for our library can be incorporated into the applications. We will later evaluate our library using some of these applications, while leaving the others as future work.

4.1 Basic Architecture

The first application to which we added support for our library is the Jetty HTTP server [22]. Jetty is a HTTP application server written in Java. It can operate either as a stand-alone program, serving either static web content or content dynamically generated by java servlets, or as an embedded web server integrated into another application. Jetty is used in several well-known software projects, including Apache Hadoop [24], Eclipse and Google AppEngine. We chose Jetty since it is integrated into many popular applications [21], which can potentially benefit from the modifications we made to Jetty. In addition, modifying Jetty allowed us to determine whether our library – which is written in C – can be used efficiently from other programming languages as well.

Figure 4.1 shows the layered architecture of a Jetty installation supporting remote IPC. In this scenario, Jetty runs atop an unmodified Java virtual machine (JVM), which in turn runs on top of the operating system (OS). Our modified version of Jetty resorts to our library for communication, making calls into the library directly from Java code. The library in turn accesses the network hardware – in our tests, we used InfiniBand (IB) – directly whenever possible, effectively bypassing the operating system for most calls.

Our implementation relies on Java Native Access (JNA) [20] for calling our

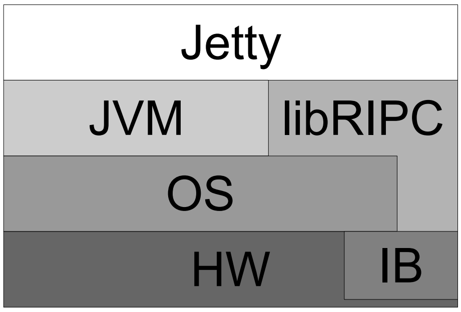


Figure 4.1: Layered architecture of an RIPC-enabled Jetty web server. Note that LibRIPC operations bypass both the JVM and the OS.

C library. JNA is slightly slower than the competing Java Native Interface (JNI) [14], but has the advantage of built-in support for C pointers, which allows us to use our library's memory management functions. JNA also supports wrapping memory referenced by a pointer into a ByteBuffer, providing direct access to that memory without copying any data. This feature allowed us to implement zero-copy communication of data even from Java code.

4.2 Integration of LibRIPC into Jetty

Jetty's modular design greatly facilitated our modifications. For example, Jetty does not access its communication sockets directly, but wraps them in so-called connector classes, several of which come with the unmodified Jetty code. Furthermore, the connector interface does not rely explicitly on sockets. The connector's main task is to accept incoming requests and forward them to a generic handler function. The connector also spawns an object for each remote party communicating with the server. That object, which is called an endpoint, is then used to communicate with the client it represents, providing a generic send/receive interface. Since the connectors and endpoints hide the backing communication channel completely from the rest of Jetty's code, we were able to implement our own connector and endpoint classes, which relies on our library instead of sockets, without modifying any other components.

Jetty further uses regular *ByteBuffers* [13] – a Java base class representing a simple buffer holding a blob of binary data – as internal storage. Jetty wraps these simple buffers in custom buffer classes which provide extra functionality. These buffers are passed to an endpoints send and receive functions as arguments, which then read data from a socket into the buffer or vice versa. Normally, these buffers are allocated by the Java virtual machine, which does not know about remote DMA or memory with special properties the network hardware may need.

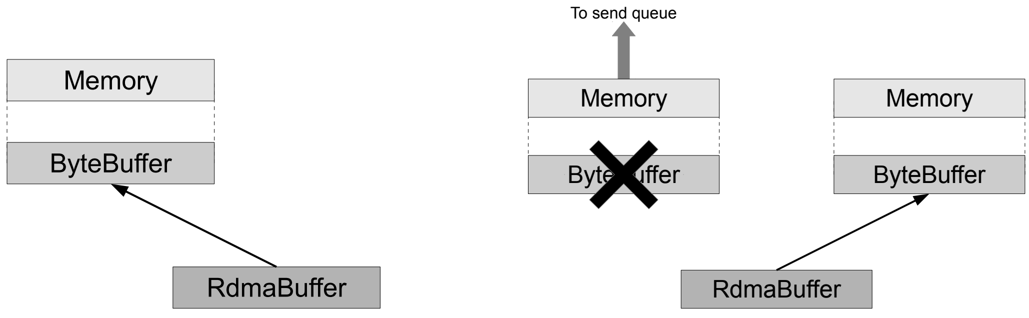
It is therefore not possible to use one of Jetty's regular buffer implementations for remote DMA without making modifications to the JVM. Instead, we implemented our own buffer class, which we called RdmaBuffer. The RdmaBuffer implementation uses our library's memory allocator to obtain a pointer to remote DMA-enabled memory outside the JVM and then wraps that memory into a ByteBuffer. The resulting buffer is compatible with Jetty's standard buffer implementation, which allows Jetty's core components to use our buffers without modification.

4.3 Buffer Handling

Unfortunately, Jetty's core classes expect sends and receives to be synchronous. For example, Jetty's code frequently expects some empty space in a buffer right after the send function it was passed to returns. However, our long send function, which is otherwise a natural fit for returning large amounts of content to a client, is not synchronous at all – it only sends a control message to the client, who can then read the data from the server's memory at some later time. The server has no way of knowing when the client will fetch the data, but if the buffer containing the data to be sent is modified before the client fetches its contents, catastrophic things might happen.

Our code therefore emulates synchronous send and receive operations by transparently swapping the ByteBuffer inside our RdmaBuffer. An example of this method applied to the sending of data is depicted in Figure 4.2 – 4.2(a) shows the buffer prior to sending, while 4.2(b) shows the situation after swapping. The data to be sent is contained in the memory backing the ByteBuffer, while the ByteBuffer acts as the backing store of the RdmaBuffer. When the RdmaBuffer is passed to the send function of our endpoint class, that function first obtains an empty, remote DMA-enabled ByteBuffer from our library's memory allocator. It then swaps the ByteBuffer backing the RdmaBuffer with the newly allocated ByteBuffer. Afterward, it passes a pointer to the data to be sent to one of the send functions of our library. While the ByteBuffer object referencing that memory is no longer needed and therefore garbage collected, the endpoint retains a pointer to the memory backing the ByteBuffer after sending and frees that memory when the next request is received from the same endpoint[1] or when no requests have been received from that endpoint for a certain time. Since Jetty – outside the connector and endpoint classes – only references the RdmaBuffer, but never the ByteBuffer backing it, that swap is completely transparent to all other modules inside Jetty – the buffer simply appears to be completely empty after the send function returns.

[1] The underlying assumption is that the client will not send another request until he has received all content from the previous one. If a client wishes to send more than one request in parallel, it needs to use a separate service ID for each request



(a) Before sending (b) After sending

Figure 4.2: Sending a data buffer from java code. The ByteBuffer backing the RdmaBuffer is swapped transparently to the application. The content of the old backing buffer is then queued for sending, while the application continues to run with an empty RdmaBuffer.

Receiving data works similarly: The endpoint's receive function swaps the backing ByteBuffer of an RdmaBuffer against a receive buffer returned by our library's receive function.

4.4 Cached Buffers

Another issue arose from Jetty's cache, which uses a different, custom buffer class. Buffers in the cache are usually never changed – on a cache hit, their content is just read into a socket without modifying the buffer. Thus, our method of swapping the backing buffer cannot be applied to cached buffers, as it would leave the buffer empty, causing clients subsequently requesting the same content to receive only null bytes. Fortunately, swapping is not necessary for cached buffers: The main reason for swapping regular buffers is avoiding modifications to their contents, but cached buffers are never modified. Thus, all that is required for proper handling of cached buffers is to omit the swapping step if the buffer is part of the cache.

4.5 Summary

The integration of our library into Jetty was surprisingly simple, even though the library was not originally meant for use by Java code. Jetty's modular design facilitated the integration of LibRIPC since it allowed us to wrap all LibRIPC-related code in a connector class, while leaving the rest of the code base unchanged. Jetty's use of ByteBuffers and custom wrapper classes around them further allowed us to implement a true zero-copy data transfer by transparently swapping buffers when needed. No modifications to the JVM were necessary, and the implementation does not assume any specific details about the network hardware.

Chapter 5

Evaluation

In this section, we will present details about how we evaluated our library. We first describe the test platform on which we conducted our experiments in Section 5.1, followed by a summary of the goals of the test process in Section 5.2. We then present some details about the prototype implementation of the library in Section 5.3, before presenting our experiments and their results in Section 5.4.

5.1 Test System

We conducted our experiments on a small, four-node InfiniBand cluster. Each of its nodes is equipped with an Intel Xeon E5520 quadcore CPU clocked at 2.27 GHz and 6 GB DDR2 RAM. Its InfiniBand network is composed of Mellanox ConnectX-2 QDR adapters connected to an HP InfiniBand DDR switch using 4x aggregated links, giving the network an effective bandwidth of 16 GBit/s. All nodes of the cluster are running a 64-bit build of CentOS 5.6 on Linux kernel 2.6.35.

5.2 Test Goals

The main goal of our experiments was to verify that our library meets the goals postulated in Section 3.2. We therefore examined the performance, overhead, ease of integration and flexibility of our library.

To measure the library's performance, we first conducted a series of microbenchmarks. We were primarily interested in the data throughput and end-to-end latency of our messages for various message sizes. During the microbenchmarks, we also monitored the test application's CPU usage to estimate the library's overhead. For comparison, we ran the same experiments using TCP sockets on top of IP over InfiniBand (IPoIB). Finally, we used Jetty [22], the web server we described in Section 4, as a real-world test case. Using Jetty, we first compared the time needed to complete HTTP requests for files of various sizes when using IPoIB and our library, respectively. Finally, we integrated Jetty into the Hadoop map/reduce framework [24] in order to evaluate Jetty under realistic load conditions.

The integration of LibRIPC into Jetty also allowed us to qualitatively evaluate the ease of integration of our library. To that end, we recorded the development time required to integrate Jetty into our library, as well as the number of code-lines modified during the integration. For comparison, we determined the number of lines of Jetty's original, socket-based code.

To determine the flexibility of our solution, we tested whether the library works as discussed in Section 3.5.1. Specifically, we tested whether servers are found by clients independent of their respective locations – including client and server running on the same machine – and whether communication between two services can seamlessly resume after a crash and relocation of the crashed service to a different node.

5.3 Prototype Implementation on InfiniBand Hardware

In this section, we will describe the implementation of our library on InfiniBand hardware. We will start with a short overview of the InfiniBand architecture, before describing the InfiniBand-specific implementation of the library's features.

5.3.1 Introduction to InfiniBand

InfiniBand is a switched-fabric interconnect featuring high bandwidth, low latency, high scalability, quality of service mechanisms and automatic failover on link failures. It is a joint effort by several large computing companies, including Compaq, HP, IBM, Intel, Microsoft and Sun. Its specifications [33] are now maintained by the InfiniBand Trade Association. InfiniBand can be used for connections between compute nodes as well as connections between compute nodes and peripherals, such as storage devices. It also supports remote DMA and atomic operations between machines. InfiniBand is widely used in high performance computing and larger datacenters. A more detailed introduction to InfiniBand and its features is given in [28]. InfiniBand network cards – in InfiniBand terms referred to as host channel adapters (HCAs) – are programmed through a set of vendor-independent *verbs*, which are specified in detail in [33].

In order to communicate using InfiniBand, each user needs to allocate *queue pairs* – consisting of a send- and a receive queue – and *completion queues*. The

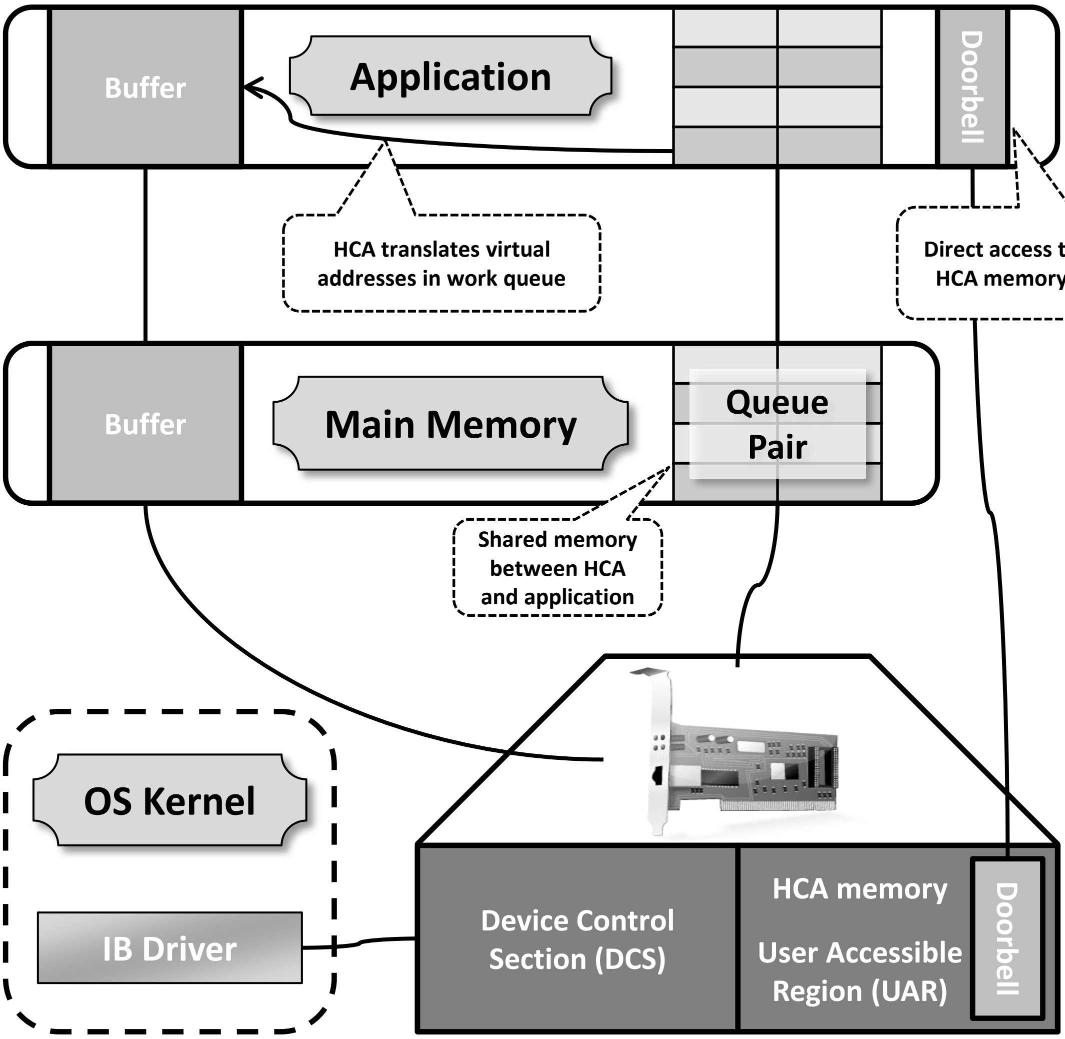

Buffer
Application
Doorbell
HCA translates virtual addresses in work queue
Direct access to HCA memory
Buffer
Main Memory
Queue Pair
Shared memory between HCA and application
OS Kernel
IB Driver
Device Control Section (DCS)
HCA memory
User Accessible Region (UAR)
Doorbell

Figure 5.1: Communication between the application and the InfiniBand host channel adapter (HCA) (Taken from [46] with permission from the author)

user then posts *work request items* representing send or receive operations into these queues. An item posted to a send queue describes either data to be sent directly or a remote DMA data transfer. Items in the receive queue describe buffers used for storing data that is received via direct send operations. Once one or more requests have been placed in a queue, the application rings a doorbell to notify the network hardware of the new work request; all further handling then is done by the network hardware. The process is illustrated in Figure 5.1.

On completion of a requested operation, the network interface generates a completion notification for that operation if requested by the user. Completion notifications are generated i) for all completed receive requests and ii) for send requests only if a completion notification was requested either for this particular send request or for all send requests using this queue pair at the time the queue pair was created. If a completion notification is generated, it is placed in the completion queue associated with the send or receive queue the work request was originally written to. Clients can check a completion queue at any time to find out if a given operation has completed yet. If requested by the user, the network hardware can also generate an interrupt when an entry is added to a completion queue, which allows threads to sleep while waiting for an operation to complete.

InfiniBand further supports various connection types. The simplest type is the unreliable datagram, which makes no guarantees with respect to lost or reordered messages, but does not require any state to be maintained. The most complex type is the reliable connection, which guarantees that all data is received in the same order it was sent, but requires a large amount of state for each connection. The amount of state required makes resource management rather complex – however, reliable connections are currently required for remote DMA.

InfiniBand hardware is capable of operating on virtual memory. It also provides fine-grained memory access control for both local and remote DMA operations. However, both of these features require the memory to be registered with the InfiniBand hardware before it can be used. Whenever a new memory buffer is registered, the relevant page table entries are copied from the operating system into the network hardware's internal page table and the memory buffer is pinned in memory to prevent swapping during a data transfer. The InfiniBand device driver also assigns two access keys per buffer on registration, one for local- and one for remote operations. The local key is needed whenever the buffer is passed as a parameter to a hardware operation – it is intended to prevent another process on the same machine from tampering with other users' buffers. The remote access key is needed to access the buffer from a remote location using remote DMA operations. The memory registration process is illustrated in Figure 5.2.

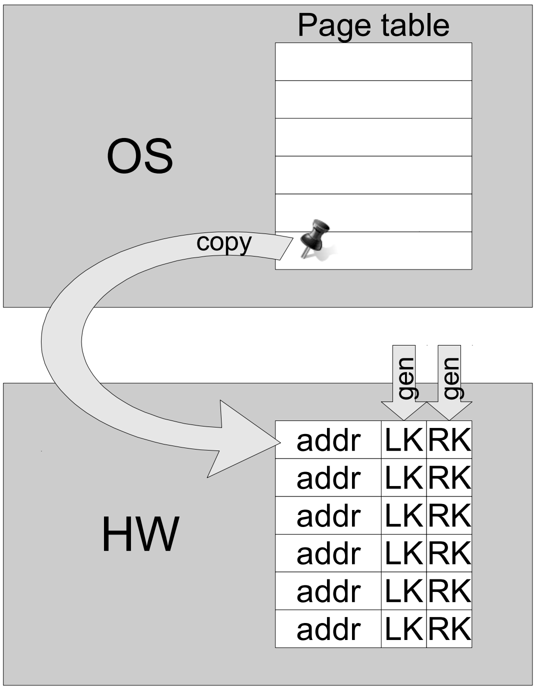


Figure 5.2: InfiniBand memory registration. The page table entry is simply copied from the OS to the network hardware. The two access keys are randomly generated by the InfiniBand device driver.

5.3.2 LibRIPC Service Contexts

In order to map from location-independent service IDs to network IDs and queue pairs, our library maintains two arrays of context items, one for locally registered service IDs and one for remote service IDs. We used static arrays to allow constant-time lookup of service information. In order to save space, the elements of both arrays are actually pointers, most of which are initialized to NULL. Only entries which are in use will instead point to a data structure holding the actual information – an idea devised from the “0-Mapping Trick” used for the TCB array in L4Ka::Pistachio [52]. The data structure describing a local service ID mainly holds references to the various InfiniBand queues used for sending and receiving data on behalf of this service. On InfiniBand, it is defined as follows:

```
struct service_id {
        uint16_t number;
        struct ibv_cq *send_cq;
        struct ibv_cq *recv_cq;
        struct ibv_qp *qp;
        struct ibv_comp_channel *cchannel;
};
```

The first member of this data structure is is the service ID itself; it is identical to its index in the array of local services. The ID is stored in the data structure to allow passing of the data structure as a pointer to a function that does not otherwise know the service's ID. The next two members are the completion queues used for send- and receive requests on the service's queue pair, respectively. The next member – qp – is the queue pair itself. The last member is a completion event channel attached to the receive completion queue. It is used to block while waiting for messages on calls to our library's receive function.

Conversely, the structure describing a remote service holds all information necessary to send messages to that service. The InfiniBand version of that data structure is defined as:

```
struct remote_context {
        uint32_t qp_num;
        struct ibv_ah *ah;
        uint32_t resolver_qp;
        enum conn_state state;
        struct ibv_qp *rdma_qp;
        struct ibv_cq *rdma_send_cq;
        struct ibv_cq *rdma_recv_cq;
        struct mem_buf_list *return_bufs;
};
```

Only the first three members of this data structure are always in use. The first member holds the number of the remote's services dedicated queue pair, to which all short- and notification messages are sent. The second member holds a so-called *address handle* of the machine hosting the remote service. As with memory buffers, remote global or local addresses we wish to send messages to must previously be registered with the hardware. The third member holds the number of the resolver's multicast queue pair, which is where remote DMA connect requests are sent.

The remaining members are used for long sends only; they are initialized to NULL until the first long send takes place. The enum called state marks the current state of the reliable connection needed for long sends (see Sections 5.3.4 and 5.3.4 for details). That state is either disconnected, connected or in the process of connecting – the latter state meaning that a connect request was sent to the receiver, but no reply has arrived yet. The next three members hold the reliable connection queue pair and the send and receive request completion queues for that queue pair, respectively. The final member is the head of a linked list of return buffers sent by the remote destination, containing, for each return buffer, the address and remote access key needed to push a reply into it.

5.3.3 Short Messages

Our library uses a single unreliable datagram queue pair to implement InfiniBand short messages. The reason we chose unreliable datagrams is that only one queue pair per local service ID is needed, and all possible senders can direct their messages to that queue pair. If we were to use a connection-oriented transport instead, each service would need one queue pair per communication partner, and the management of these queue pairs would increase the overhead of the library.

When a new service ID is allocated, the library first creates a queue pair and the corresponding completion queues for that service ID. It then puts a fixed number of receive buffers into the receive queue. The size of these buffers is equal to the hardware MTU, which can be queried from the network device. Note that allocating larger buffers makes no sense, as unreliable datagrams larger than the MTU are silently dropped by the hardware. The number of receive buffers determines how many messages can be queued if the frequency of arriving messages is higher than the frequency of receive calls by the application. Messages for which no receive buffer is available upon arrival are silently dropped.

In order to send a short message, the user passes an array of pointers to the library. This array is directly transformed into a scatter-gather-list, which is then passed to the hardware as part of the send request. The hardware then fetches the data from the user-specified locations – there is no need to copy all parts of the message into a single buffer first. On the receiver side, all parts of the message appear in one receive buffer. From the application's point of view, that receive buffer is ordinary memory, so a pointer to the received message can be passed to the function that called receive – there is again no need to copy the data. Also note that the hardware automatically removes each buffer from the receive queue after it has been used, so the message is not accidentally overwritten by a subsequent message.

5.3.4 Long Messages

The concept of long messages introduced in Section 3.4.2 maps well to InfiniBand. The only major issue is the mandatory registration of memory – the user must place the message into a registered buffer before it can be sent using remote

DMA. Our library provides a function that allocates a pre-registered buffer, which can be used like any memory buffer obtained by a call to `malloc()`. There is also a function that registers an already existing buffer, but since InfiniBand hardware can only handle whole pages, memory adjacent to the buffer may be registered unintentionally if the buffer is not aligned to the page size. The library's own allocation function automatically aligns the memory correctly; it is therefore advisable to always allocate pre-registered buffers for any data that might be sent as a long message.

Once a buffer has been registered, any part of the buffer may be passed to our long send function by specifying its starting address and size. In contrast to the buffer itself, the starting address need not be aligned to a page boundary. The library then creates a long send descriptor item and sends it to the receiver. That descriptor item is hardware-specific; for InfiniBand, its layout is as follows:

```
struct long_desc {
        uint64_t addr;
        size_t length;
        uint32_t rkey;
        uint8_t transferred;
};
```

The addr field of this descriptor contains the virtual starting address of the message's payload in the sender's address space, while the length field contains the message's size. The rkey field holds the remote access key of the buffer, which is assigned by the InfiniBand hardware on registration of the buffer. The transferred field is used for the return buffer optimization – it indicates whether the contents of the message have already been written to a return buffer when the descriptor item is received.

5.3.5 Remote DMA Connection Establishment

Prior to using remote DMA operations on InfiniBand, the sender and receiver need to establish a connection using reliable connection queue pairs. In order to establish that connection, each side must configure its queue pair using information from the other side. Thus, a handshake between the communication partners is required to exchange that information.

Figure 5.3 depicts the process of establishing the connection. First, the process initiating the connection creates a reliable connection queue pair and configures it with all necessary information known locally, such as initial sequence numbers. Then, it sends a connect request to the remote party, containing all information necessary to fully configure the queue pair on the remote end. That message is received in a separate thread in the receiving process in order not to disturb other

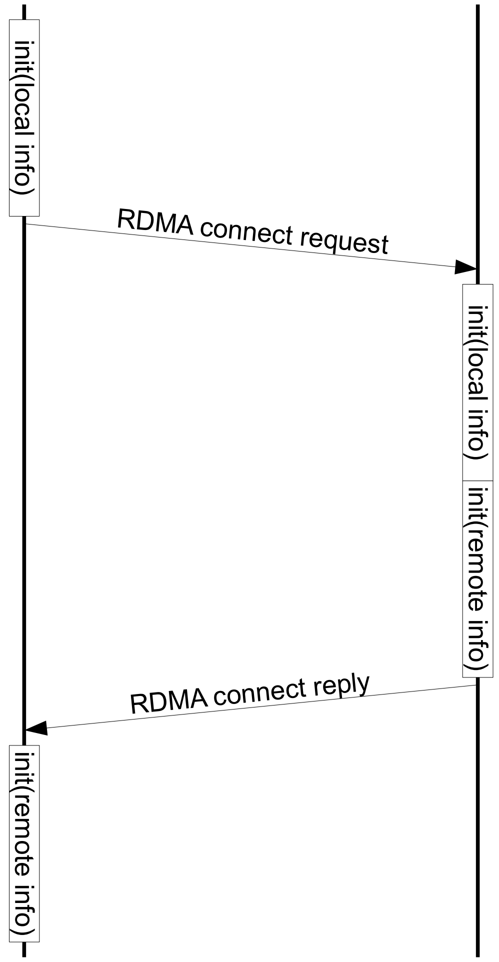


Figure 5.3: Remote DMA connection establishment. The process on the left has to wait for the process on the right before it can finish its own initialization.

services on the same node. Upon reception of the connect request, that thread first configures a queue pair for the service the sender wishes to connect to. It then sends a reply containing all information the requesting service requires to finish its own initialization. When the initialization is complete, both services can send long messages to the other side of the connection as described in Section 5.3.4.

On InfiniBand, the LibRIPC's handshake messages use the following layout:

```
struct rdma_connect_msg {
        enum msg_type type;
        uint16_t dest_service_id;
        uint16_t src_service_id;
        uint16_t lid;
        uint32_t qpn;
        uint32_t psn;
        uint32_t response_qpn;
};
```

Like most other message types, the handshake messages start with a `type` field, followed by the service IDs of the destination and the origin of the connection request. The `lid` field holds the local ID, a subnet-local address of the node hosting the sending service. Both ends of the connection use this field to send their own local ID to their respective partner. The `qpn` field holds the number of the reliable connection queue pair the sender has allocated for this connection. The `psn` field holds an initial sequence number, which on both sides is initialized to a random number. Each side of a connection must know the current sequence number of the other side, as packets with an unexpected sequence number are automatically discarded. The purpose of the `response_qpn` field is merely ease of implementation: Since InfiniBand can manage up to 2^{24} queue pairs, we currently use separate queue pairs to distinguish response messages from other connection requests. The `response_qpn` field holds the queue pair number the response to the current connection request is to be sent to. On architectures where resources are more constrained, requests and responses may use the same resources; in that case, the `response_qpn` field is not necessary.

5.3.6 The Resolver

Our library currently uses a multicast-based resolver to obtain hardware-specific addressing information from service IDs. That resolver runs as a separate thread in each library instance. InfiniBand features efficient distribution of multicast messages. However, before multicast can be used, a complex setup of *multicast groups* is required. Multicast groups are identified by a unique *global ID*, a globally routable address also used to reference machines in other subnets. Once a multicast group has been set up, any machine can join that group. Messages to the group's global ID are then received by all members of the group.

Before a node can receive any multicast traffic, it must attach one or more of its network ports to one or more multicast groups. Attaching is done by sending a special network packet, called *Management Datagram*, to the *Subnet Manager*. The Subnet manager is a software program responsible for assigning addresses and configuring routing in an InfiniBand subnet. There is generally no need to know the exact location of the subnet manager – all information necessary to reach it can be queried from the local network hardware. The user is free to choose the global ID of the multicast group to attach to – if a group with the chosen ID does not exist, it is implicitly created. If the attachment request is successful, the next step is to attach one or more unreliable datagram queue pairs to the multicast group, which is done locally – that is, without involving the subnet manager – using the verbs interface. Queue pairs need not be attached to the group in order to send to it: Any message sent to the multicast group's global ID using a defined, well-known queue pair number will be received by all queue pairs attached to the group.

When an instance of our library starts its resolver thread, it first tries to attach to a well-known multicast group used for all resolver messages. The global ID of that group is known to all library instances, so no further configuration is necessary. It then prepares two queue pairs; one of these queue pairs is later used to receive resolver requests, while the other is used to send replies. The reason for using two queue pairs lies in the multithreaded nature of our resolver: Using two queue pairs, an application thread sending a resolver request can block on one queue pair, while our dedicated resolver thread can simultaneously receive requests on the other one. The resolver then attaches the first queue pair to the multicast group and waits for incoming messages.

Once a message has been received, the resolver first checks if the requested service ID exists in the resolver's library context. If so, it sends a response message containing all information necessary to contact the requested service to the requester. The resolver also updates its own cache with a description of the requesting service using the information contained in the resolver message. The process of resolving a service ID is illustrated in Figure 5.4.

Requesting information about a service ID is straightforward: The library prepares an appropriate resolver message and sends it to the well-known global ID of the resolver's multicast group. On InfiniBand, the resolver messages – both requests and replies – have the following format:

```
struct resolver_msg {
        enum msg_type type;
        uint16_t dest_service_id;
        uint16_t src_service_id;
        uint16_t lid;
        uint32_t service_qpn;
        uint32_t response_qpn;
        uint32_t resolver_qpn;
};
```

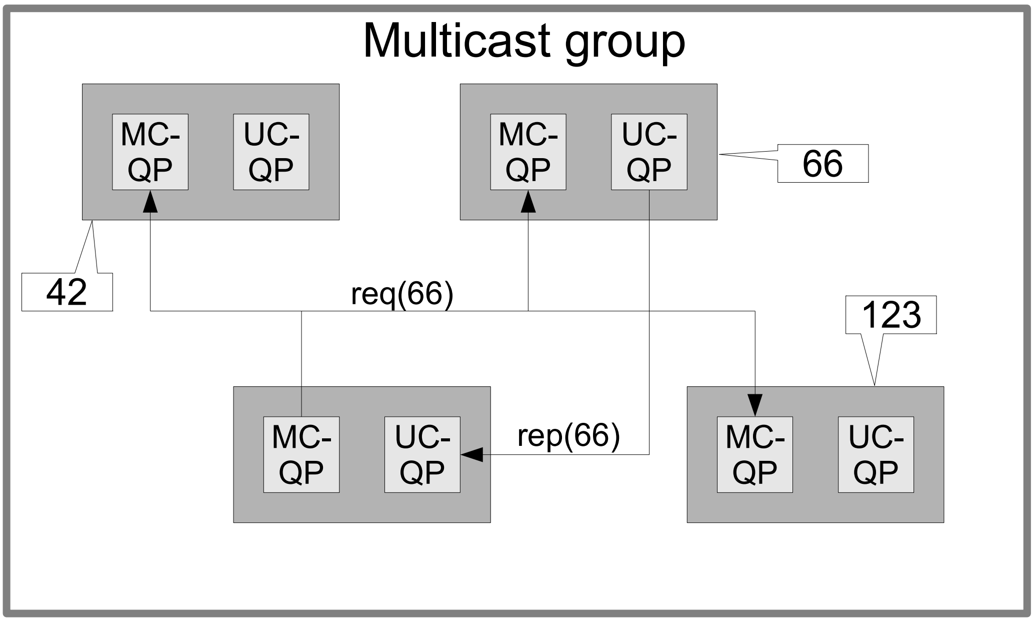


Figure 5.4: Resolving information about a service ID. The request is received by the multicast queue pairs of all resolvers present in the network. The resolver thread of the process containing the requested service ID then replies via unicast.

The first three fields of the message denote the message type, the requested service ID (`dest_service_id`) and the ID of the service sending the message (`src_service_id`), respectively. The `lid` field contains the local ID of the machine hosting the requesting service, while the `service_qpn` field contains the queue pair number assigned to the requesting service. Using those two fields, all resolvers receiving the message update the requesting service's remote context, which potentially saves additional multicast messages if other services wish to communicate with the requesting service later on. The `response_qpn` field holds the number of the requesting resolver's unicast queue pair, which is where the response to resolver requests should be sent. The `resolver_qpn` field contains the number of the queue pair used to establish remote DMA connections (see Section 5.3.5). Currently, the resolver's multicast queue pair is used for that purpose, but a different queue pair could be used in its place.

5.3.7 Buffer Management

As mentioned in Section 5.3.1, the memory registration needed for InfiniBand can be costly if used frequently. Our library therefore tries to re-use already registered buffers whenever possible in order to reduce the registration of new memory buffers to a minimum. To achieve this, the library maintains two linked lists of memory buffers: A list of buffers that are currently available and a list of buffers currently in use. The use of the first list is straightforward: Whenever the user – or some internal function of the library – wants to allocate memory that can be used with the network hardware, the library first checks the free list if a large enough buffer is currently available. If that is the case, that buffer is moved to the list of used buffers and a pointer to the buffer is returned. If no buffer is currently available, or all available buffers are smaller than needed, a new buffer is allocated, registered with the hardware, inserted in the list of used buffers and then returned to the caller.

The purpose of the used buffer list is mainly to hide the details of the network hardware from the user. It holds the information returned by the registration function, most notably the local and remote access keys. The used buffer list is only used by internal library functions to retrieve the registration information belonging to pointers that were passed by the user. That way, the user can work with library buffers as he would with any buffer returned by `malloc()`, while the library can look up the other related information if necessary.

We observed that the library often issues multiple consecutive lookups for the same entry in the used list. This happens, for example, when the user allocates a buffer directly prior to sending a message or if one buffer is used to hold multiple items of the same message. We therefore arranged the used buffer list as a stack rather than a first-in/first-out list, which ensures that recently used buffers are near the top of the list. As a result, most lookups only need to traverse the first few items.

5.4 Experimental Results

In this section, we will describe the experiments we conducted as well as their results. We will first review its performance in Section 5.4.1, before examining its flexibility in Section 5.4.2, and its ease of integration in Section 5.4.3.

5.4.1 Performance

Ping-pong Test

We first conducted a simple ping-pong-type microbenchmark. In this experiment, we started two services on two different machines. The two services then exchanged small messages in a tight loop, with each of them waiting for a message from the other service before sending the next message. We measured the time needed to exchange 10000 messages, and then divided that time by 10000 to ob-

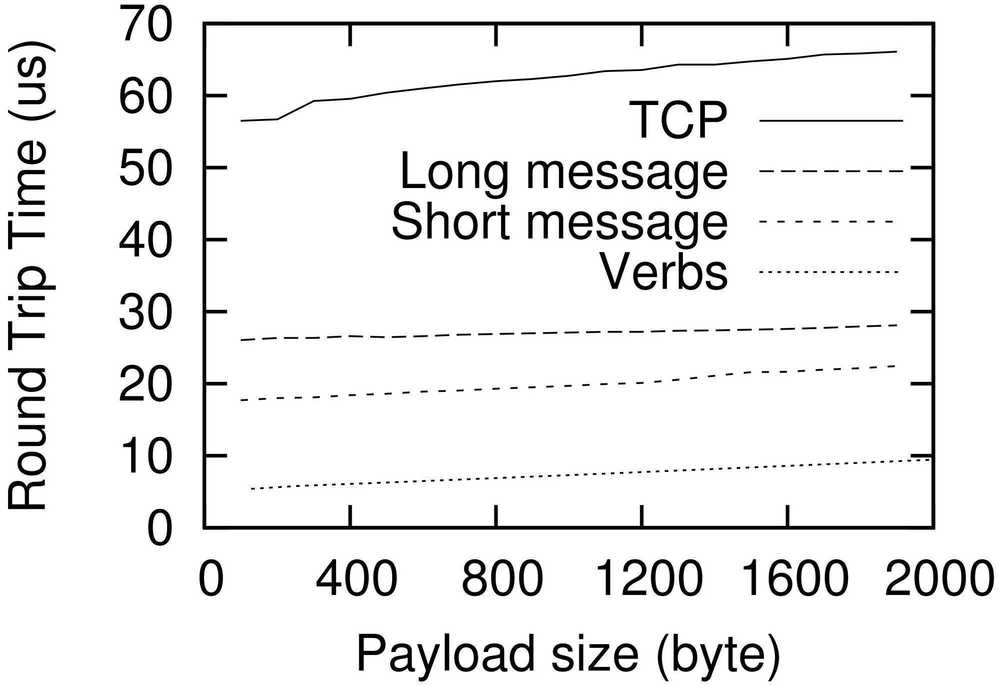


Figure 5.5: Pingpong latencies of short- and long messages compared to TCP sockets

tain the average round-trip time. We repeated this experiment using short- and long messages and, for comparison, a TCP socket on top of IP over InfiniBand, using message sizes ranging from 100 to 1900 bytes in each test.

Figure 5.5 shows the results of this first experiment. Not surprisingly, the latency-optimized short messages achieved the best round-trip time, ranging from 17.72 (100 bytes) to 22.44 (1900 bytes) microseconds. As expected, the latency is slightly higher than that of raw InfiniBand verbs – which in our measurements ranged from 5.38 (128 bytes) to 9,54 (2048 bytes) microseconds – but still reasonably low. The latency of long messages was only slightly worse, ranging from 26.07 (100 bytes) to 28.09 (1900 bytes) μs. In addition, the CPU load during the experiment was generally below 1% for both short- and long messages. As expected, the round-trip time using sockets was a lot worse, ranging from 56.5 (100 bytes) to 66.11 (1900 bytes) μs. In addition, the CPU load increased to 4 – 5 % when using sockets. The large difference between sockets and our library most likely stems from the additional system calls and copying of data between user- and kernel-level, which our library does not require.

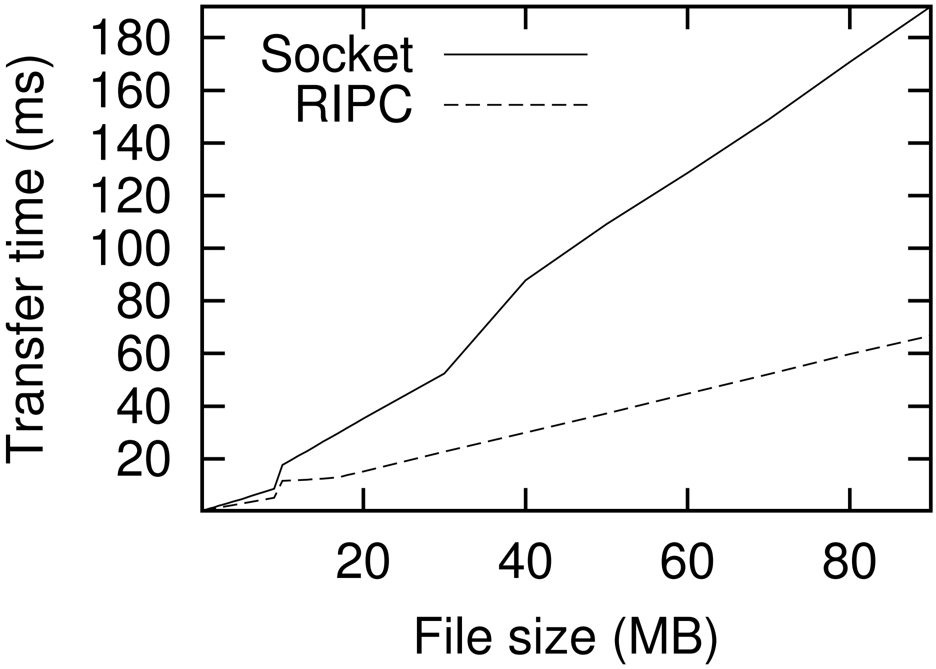
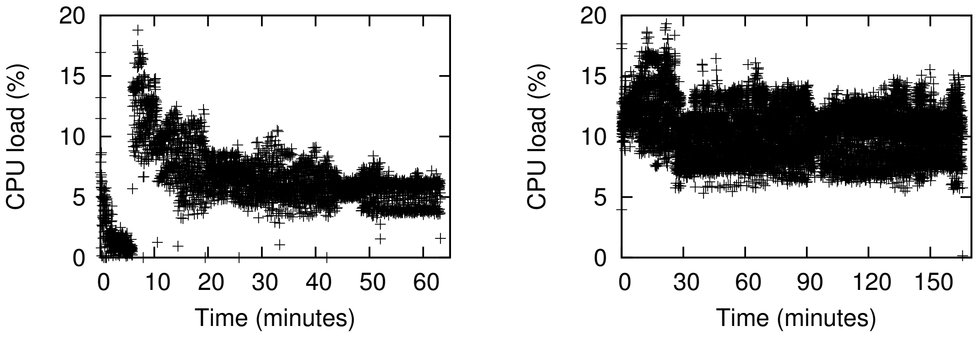


Figure 5.6: Turnaround time of HTTP requests using Jetty

HTTP Request Latency

In the second experiment we conducted, we used the Jetty web server we described in Section 4.1. We created a test program which requests files of various sizes from a Jetty instance running on a different machine. The test program requests the files using short messages and then waits for the file to arrive as a single long message before sending the next request to the web server. Again, we repeated this procedure 10000 times and then obtained the average turnaround time. We used files filled with pseudo-random data, with file sizes ranging from 100 KB to 90 MB. For comparison, created a second test program, which sends the same requests as the first one using a TCP socket on top of IP over InfiniBand instead of LibRIPC. That second test application makes use of libcurl [19], which implements the HTTP requests efficiently, for example by re-using active sockets for sending subsequent requests to the same server.

Figure 5.6 shows the results. The plot shows that Jetty achieves a lower latency using our library in almost any case. For large files, the total turnaround time is dominated by the actual data transfer, which leads to our library's remote DMA-based data transfer outperforming the socket-based alternative by far. However, for files smaller than approximately 700 KB, socket-based communication



(a) RIPC

(b) Socket

Figure 5.7: CPU load of a Jetty web server servicing a large number of requests

is slightly faster than our library. We believe this is due to the overhead incurred by calling our library – which is written in C – from Java code and the initial setup cost of the remote DMA data transfer, which becomes more important as the message payload grows smaller.

In order to estimate the overhead of our current implementation, we also measured Jetty's CPU load while running the test described above. We used sysstat [25] to measure the test system's overall CPU load, collecting one data point every second. The results are shown in Figure 5.7. Specifically, the plot shows the sum of kernel and user load over the course of the experiment.

Using our library (Figure 5.7(a)), the CPU load is initially low. During that time – roughly the first six minutes –, the test application requests small files, which are served from Jetty's internal cache. Since Jetty keeps cached data in RAM, that data can be sent directly from the cache, without any need for copying. Furthermore, the data is ultimately transferred using a remote DMA operation, which is executed without CPU involvement. After six minutes, the files being requested become larger than the configured maximum size of cached files. At that point, the CPU load increases significantly, briefly surpassing the CPU load when using sockets. The reason for this high load is that, Without the cache, the files being requested are read from the hard drive on every request, which also involves allocation of resources such as buffers. However, that overhead is eventually offset by the lower CPU impact of the remote DMA transfer as the file size further increases. Starting around the 10 minute mark, the CPU load starts to flatten out to around 5% for large files.

We found the CPU load to be somewhat higher when we repeated the same experiment using a TCP socket on top of IP over InfiniBand (Figure 5.7(b)). Even for files served from the cache, the CPU load was constantly above 10%. As in the previous experiment, the load briefly increased as files grew larger than the caching threshold after around 10 minutes and then flattened out around 10%. Overall, these results show that the offloading implemented in our library decreases CPU load in the majority of cases, especially when the data to be sent is already present in RAM.

Hadoop

To further evaluate our modified version of Jetty, we integrated it into Apache Hadoop [24] version 1.0.0. Hadoop is an open-source framework for the map/reduce programming model [16]. Map/reduce programs consist of a set of *map tasks* and a set of *reduce tasks*; these tasks need not execute on the same machine, but can instead be spread across a cluster of compute nodes. Each map task processes a chunk of input data and produces an intermediate output. The reduce tasks then combine that intermediate output to produce the final program output.

Hadoop uses Jetty – among other things – to exchange data between map and reduce tasks; a reduce task requests a map task's output by sending a simple HTTP request to the compute node the map task ran on. We chose the data path between map and reduce tasks as a benchmark not only because large data payloads are exchanged on that path, but also because the exchange of that data takes place in parallel with the computation of other tasks. We therefore believe that Hadoop tasks will not only benefit from LibRIPC's high throughput, but that its low overhead will also allow those tasks to finish their computation faster.

In order to integrate our modified Jetty web server into Hadoop, we modified Hadoop's HTTP client code to use LibRIPC instead of sockets for HTTP communication. Our current version of Hadoop uses short messages to send HTTP requests between map and reduce tasks. For the replies, we used long messages, sending the entire output of each map task in a single message. Note that we made no other modifications to Hadoop outside the communication path between map and reduce tasks. All other parts of Hadoop therefore still use socket-based communication. One major obstacle during the integration process was Hadoop's internal data handling: In contrast to Jetty, which uses `ByteBuffers` internally, Hadoop uses ordinary arrays of `byte` to store data. However, conversion of data received from C code to a Java `byte` array requires copying the data. Our current version of Hadoop therefore can not take full advantage of LibRIPC's zero-copy semantics without extensive modifications to its core.

We evaluated the performance of our modified Hadoop cluster using the TeraSort benchmark [23] suite shipped with Hadoop. In our tests, we used two compute nodes, which together ran 8 map- and 8 reduce tasks, and a 1 GB data set. Since Hadoop does not start any reduce tasks until some map output is ready for them to process, we limited the number of map tasks per compute node to one. That limitation ensures that the reduce tasks start copying data while computation

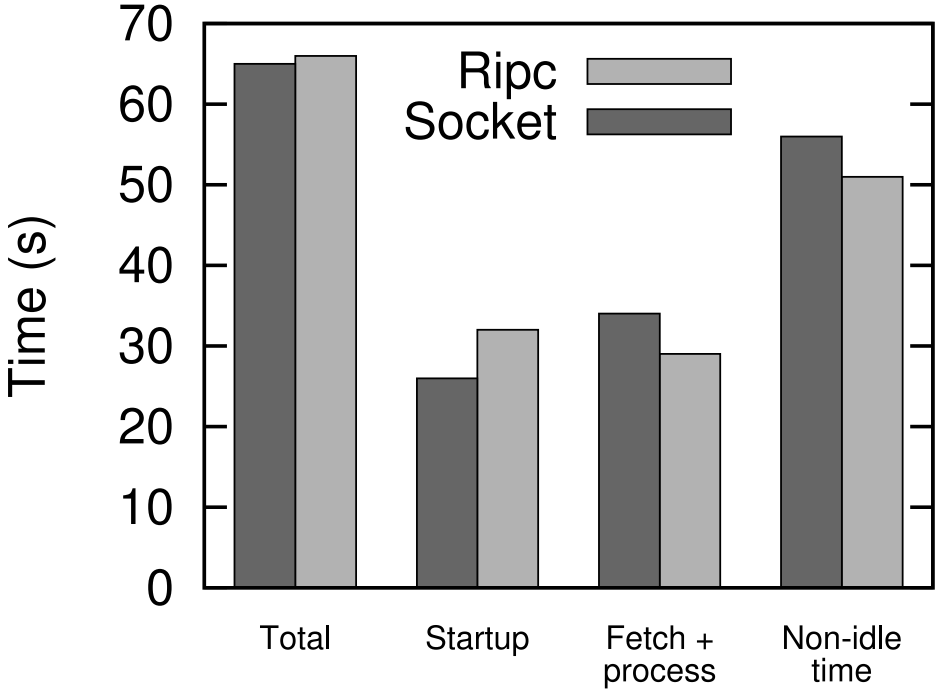


Figure 5.8: Hadoop performance results. From left to right: The total time needed for our test job, the accumulated CPU time spent in all the reduce tasks, the startup latency of the reducers and the time it took the reducers to finish after starting up.

is still in progress, thus allowing us to examine the interaction of map and reduce tasks. For comparison, we repeated the same experiment using an unmodified Hadoop installation, which uses IP over InfiniBand for communication between map and reduce tasks.

Figure 5.8 shows the average results of 10 test runs for each version of Hadoop. As can be seen from the two leftmost bars labeled “Total”, our current implementation does not yet achieve a speedup compared to the unmodified version of Hadoop. Specifically, it took our modified version of Hadoop 66.7 seconds on average to finish our test job, while the unmodified version finished in 65.5 seconds on average.

To investigate Hadoop’s slowdown when using LibRIPC, we measured the initialization time of the reduce tasks, which we defined as the time from starting the experiment to the first time a reduce task reports progress to the Hadoop framework. We found the initialization time – shown in the pair of bars labeled “startup” – using LibRIPC (32 seconds) to be significantly higher than using sockets (26.2 seconds). We believe the allocation of service IDs to be responsible for this increase in initialization time – each reduce task allocates multiple service IDs, each of which requires several resource- and buffer allocations. We also measured the actual running time of the reduce tasks after initialization – that is, the time from the first progress report of a reduce task to the end of the experiment. That running time includes the transfer of the map outputs to the reduce tasks as well as any processing required for combining the map outputs into the final program output. The results, which are shown in the pair of bars labeled “Fetch + process”, show that the run time using LibRIPC (29.1 seconds) is slightly shorter than the run time using sockets (34.3 seconds). However, it is important to remember that the run time includes a significant amount of data processing, which does not benefit from our optimizations at all.

Finally, we evaluated the CPU overhead associated with the communication. To that end, we measured the amount of CPU time – that is, the time during which the reduce tasks’ CPUs were not idle – spent by all reduce tasks combined. That time includes all processing needed to combine the map outputs into the final program output as well as any processing time spent for communication. The results are shown as the pair of bars labeled “Non-idle time”. While the unmodified Hadoop installation averaged 56,97 seconds, our modifications reduced the CPU time to 51,37 seconds, indicating that LibRIPC’s lower overhead does indeed free up CPU time for computation.

The lower CPU load and shorter finish time when using LibRIPC indicate that Hadoop does indeed benefit from LibRIPC’s high throughput and low overhead. However, since our current measurements include a large amount of data processing which is not affected by our work, we do not yet know how significant that benefit really is. We therefore plan to perform more detailed measurements of only the communication phase of the reduce tasks in the near future. We are also investigating more communication-centric benchmarks as replacements for TeraSort. In addition, the high setup cost of the reduce tasks degrades performance, especially for small jobs. Since that setup cost is constant, we believe that using more map and reduce tasks and/or larger amounts of input data would offset the setup cost: Using larger amounts of data would increase the benefit of LibRIPC’s high throughput, while more tasks would lead to more parallelism of computation and communication, in which case LibRIPC’s lower overhead would be beneficial. Unfortunately, both larger numbers of tasks and larger input data are still problematic with our current implementation, but we hope to solve these problems in the near future.

5.4.2 Flexibility

Since Hadoop was designed with fault-tolerance in mind, it also proved to be an excellent testing ground for LibRIPC’s flexibility. Hadoop not only starts and stops processes frequently, but also constantly monitors all its processes for failures. If a process fails, it is terminated and restarted on a different compute node. However, since the process is essentially the same task as before – specifically, it is assigned the same internal ID and performs the same work – it is assigned the same service IDs as before. From the other tasks' points of view, those service IDs therefore appear to move from one compute node to another.

We found that LibRIPC did provide the amount of flexibility needed by Hadoop. Newly created service IDs could join the network at any time and start communicating instantly; all necessary details – such as hardware addresses – were resolved smoothly on demand. Our library also handled process restarts so well that we did not need additional failure detection mechanisms in Hadoop's code. Since the resolver cache of a freshly started process is always empty, the restarted processes first sent resolver broadcasts when they were ready to resume communication. Upon reception of these broadcasts, the other processes updated their resolver caches with the information contained in the broadcast, thus learning the new locations of the moved service IDs. Communication then resumed seamlessly.

5.4.3 Ease of Integration

One of our goals was the ability to easily integrate our library into applications. While difficulty is hard to measure, there are metrics which at least provide an estimate of the development effort required. One such metric is the number of lines of code required to integrate support for our library into an application.

Figure 5.9 shows the number of code lines of the client applications we used for the experiments described in Section 5.4.1. "Pingpong" refers to the sum of both applications (client and server) used in the pingpong test, while "HTTP client" only refers to the client applications used to connect to the Jetty web server. In both cases, we only counted the lines of code in the client applications themselves, without including any libraries or header files they include. All line counts were measured using David A. Wheeler's SLOCCount utility [66].

As it turns out, our test applications in fact tend to be larger when using our library instead of sockets. The difference is especially high for the pingpong application. The difference stems from the fact that the library application needs to initialize the library and allocate library buffers before communicating, which is not necessary for the plain socket implementation. For the HTTP client application, however, the difference is negligible (1 line), as the socket version of that application also has a library – namely libcurl – to initialize.

We also compared the size of the various connector classes included in our final version of Jetty. Again, we used SLOCCount to measure the size of the various class files. Since some connectors are composed of more than one class, we included classes used by some but not all three connectors in the final counts. The

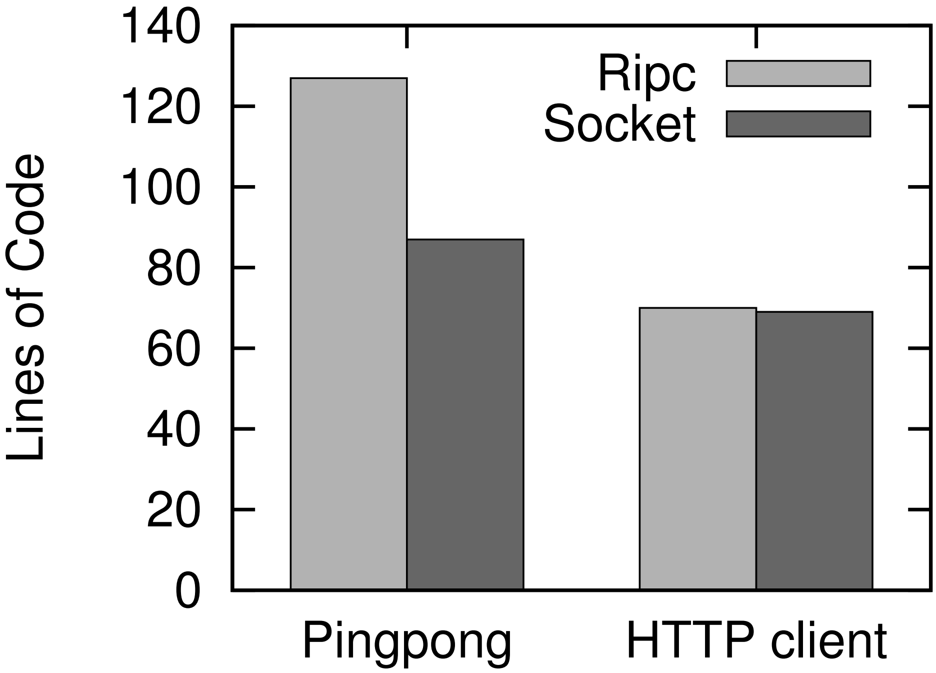


Figure 5.9: Number of code lines for simple client applications

results are shown in Figure 5.10. As you can see, the SelectChannelConnector, which is the class Jetty uses by default, is much larger than our newly written connector, spanning three core classes comprising 1724 lines of code. However, our connector – which, including dependencies, comprises 683 lines of code – is larger than the SocketConnector, which – comprising only 261 lines of code – is the smallest connector class included in Jetty. We attribute the difference in size to two factors. First, the SocketConnector is indeed very simple. It does not access any of Java's performance-related socket features, but uses only blocking I/O in its simplest form. Second, our connector class includes a fair amount of marshalling and unmarshalling code, which is necessary to bridge the gap between C and Java. For example, it is necessary to manually serialize arrays of pointers in memory before passing them to our library. This extra work is not necessary for socket-based connectors.

It is also noteworthy that we were able to finish a first, working version of our new connector within only one week. The biggest challenge we faced while developing the connector was the integration of C and Java code; using the library seemed otherwise straightforward. This leads us to believe that the difficulty of using our library is sufficiently low, though it may be slightly higher than the difficulty of socket programming.

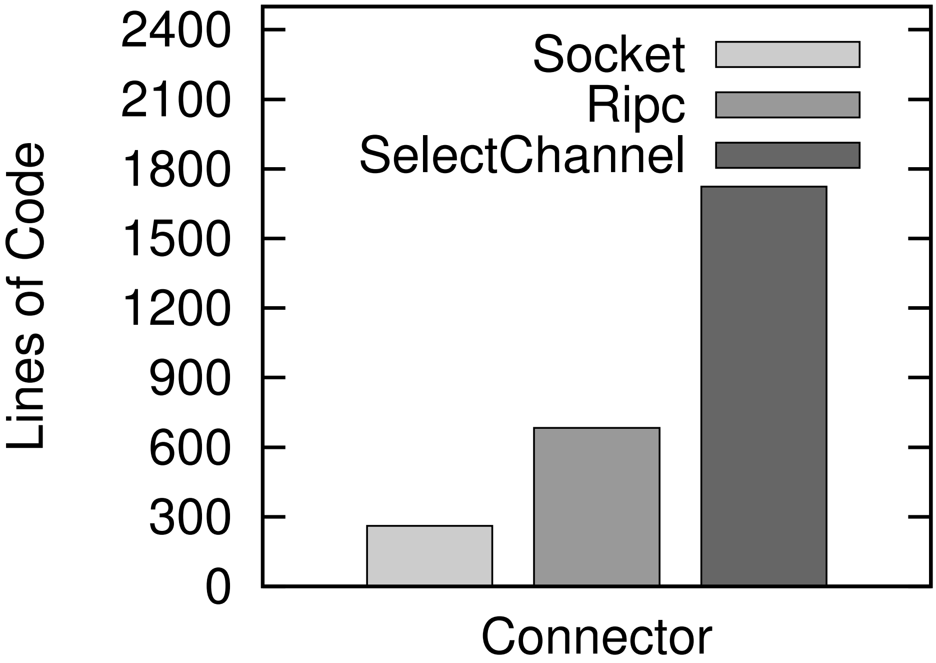


Figure 5.10: Number of code lines for our RipcConnector compared to two of Jetty's default connector classes

5.4.4 Summary

Our results indicate that LibRIPC meets the goals postulated in Section 3.2. It delivers higher throughput, lower latency and lower overhead than TCP/IP-based communication. It also provides the necessary flexibility for cloud applications by supporting starting, stopping and restarting of processes. The only slight drawback lies in the usability of LibRIPC: Programs using LibRIPC tend to grow slightly larger than their socket-based counterparts; however, neither the code size nor the development effort are completely out of proportion. We also found LibRIPC to be sometimes difficult to integrate into existing applications. Some applications – such as Hadoop – require extensive modifications to their code base in order to fully benefit from LibRIPC's zero-copy semantics. At least for newly written applications, however, we believe the difficulty of using LibRIPC to be sufficiently low.

Chapter 6

Conclusion

In this thesis, we have shown that network performance in cloud systems using high performance interconnects can be improved without sacrificing flexibility and ease of use. To that end, we have presented LibRIPC, a light-weight communication library which provides a hardware-independent interface to such interconnects. LibRIPC offers a message-based interface, providing separate send function for control- and data traffic. It uses hardware features like user-level I/O and remote DMA whenever possible to improve throughput and latency and reduces associated overhead to a minimum by caching the results of expensive operations. Furthermore, the library provides the flexibility needed by cloud applications by using hardware- and location-independent addresses called service IDs, which are resolved to hardware-specific addresses on demand.

We conducted an initial evaluation of LibRIPC using the Java-based Jetty web server on InfiniBand hardware. We integrated Jetty into the Hadoop map/reduce framework in order to create realistic load conditions. Our results indicate that both throughput and latency are improved substantially compared to TCP/IP networking. Due to the hardware's offloading features, the CPU load is improved as well. Since Jetty was originally written using Java sockets, some modifications to its code base were required – however, the complexity of these modifications proved to be manageable. This leads us to believe that the difficulty of using LibRIPC is not significantly higher than that of socket programming.

Our results verify our initial claim that sockets are not an optimal abstraction for high performance interconnects. We have shown that network performance can be improved significantly, yet without unduly increasing application complexity, by using an interface with a slightly lower abstraction level than sockets. Virtually all past work on making the performance of high-performance interconnects available to general-purpose applications has focused on improving sockets – yet there is much room for improvement by considering different interfaces. It is our hope that this work will inspire further research in that direction.

6.1 Future Work

So far, Jetty is the only real-world application we have ported to LibRIPC. While the results were significant, it remains to be seen if different workloads can benefit equally from using LibRIPC instead of sockets. Therefore, our foremost goal is to port more applications to LibRIPC and study their performance. MongoDB, which we already investigated theoretically, is a likely target; others include Redis [57] and Zookeeper [32].

An equally interesting question is whether LibRIPC performs equally well on network hardware other than InfiniBand. Currently, a port to BlueGene is underway and we are looking into iWARP [29] and RoCE [12] as additional targets. Another option is an implementation using TCP or UDP as a backend, though such an implementation would probably not perform as well due to its lack of hardware acceleration.

As soon as multiple hardware backends are available, we also plan to add support for multiple backends to be active concurrently. The mechanism we envision is similar to FABLE [58]: For each message, LibRIPC dynamically selects the fastest of multiple available communication paths. Such a mechanism would allow for efficient hosting of LibRIPC-based applications even in heterogeneous environments.

To date, the asynchronous notifications described in Section 3.4.3 are still unimplemented. We therefore plan to add that feature to our prototype and to evaluate its benefit to applications. Another desirable feature is support for group communication. Many high-performance network architectures support some form of broad- or multicast in hardware; we plan to integrate support for these hardware features into a future version of LibRIPC by supporting a notion of group service IDs, which can be used in send and receive operations like regular service IDs, but deliver messages to all members of the groups.

We also plan to investigate different solutions for process naming and name resolution. We expect our current, broadcast-based resolver not to scale well, and we therefore plan to integrate a solution using a central directory service like Zookeeper [32], which would completely eliminate the broadcast traffic. Zookeeper itself was designed with scalability in mind and thus should allow LibRIPC to scale to large platforms. Using a central repository could also allow us to propagate changes, such as crashed or migrated services, to other nodes more efficiently.

Bibliography

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

[2] Cristina L. Abad and Rafael I. Bonilla. An analysis on the schemes for detecting and preventing ARP cache poisoning attacks. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, pages 60–67, Washington, DC, USA, 2007. IEEE Computer Society. http://dl.acm.org/citation.cfm?id=1270388.1270966.

[3] Jonathan Appavoo, Marc A. Auslander, Maria A. Butrico, Dilma Da Silva, Orran Krieger, Mark F. Mergen, Michal Ostrowski, Bryan S. Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–441, 2005.

[4] Jonathan Appavoo, Volkmar Uhlig, Jan Stoess, Amos Waterland, Bryan Rosenburg, Robert Wisniewski, Dilma Da Silva, Eric van Hensbergen, and Udo Steinberg. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 385–394, Chicago, IL, USA, June 2010.

[5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol modifications for the dns security extensions. RFC 4035 (Proposed Standard), March 2005. http://www.ietf.org/rfc/rfc4035.txt. Updated by RFCs 4470, 6014.

[6] Scott Atchley, David Dillow, Galen Shipman, Patrick Geoffray, Jeffrey M. Squyres, George Bosilca, and Ronald Minnich. The common communication interface (CCI). In *Proceedings of the 19th IEEE Symposium on High Performance Interconnects*, HOTI '11, pages 51–60, Washington, DC, USA, 2011. IEEE Computer Society. http://dx.doi.org/10.1109/HOTI.2011.17.

[7] Olivier Aumage, Élisabeth Brunet, Nathalie Furmento, and Raymond Namyst. NewMadeleine: a fast communication scheduling engine for high performance networks. In *CAC 2007: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007*, pages 1–8, Long Beach, California, USA, March 2007. IEEE Computer Society Press. `http://hal.inria.fr/inria-00127356`. Also available as LaBRI Report 1421-07 and INRIA RR-6085.

[8] Luiz André Barroso and Urs Hölzle. *The datacenter as a computer: An introduction to the design of warehouse-scale machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, November 2009.

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the 22th Symposium on Operating System Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. `http://doi.acm.org/10.1145/1629575.1629579`.

[10] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. `http://dl.acm.org/citation.cfm?id=1298455.1298487`.

[11] Calxeda. EnergyCore processors. `http://www.calxeda.com/products/energycore`.

[12] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. 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*, pages 123–130, New York, NY, August 2009.

[13] Oracle Corporation. Bytebuffer (java platform se 6). `http://docs.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html`.

[14] Oracle Corporation. Java se 7 java native interface-related apis and developer guides. `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/index.html`.

[15] Dennis Dalessandro and Pete Wyckoff. Memory management strategies for data serving with RDMA. In *Proceedings of the 15th IEEE Symposium on*

High Performance Interconnects, HOTI '07, pages 135–142, Washington, DC, USA, 2007. IEEE Computer Society. `http://dx.doi.org/10.1109/HOTI.2007.21`.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008. `http://doi.acm.org/10.1145/1327452.1327492`.

[17] M. Dempsky and OpenDNS, Inc. DNSCurve: Link-level security for the domain name system. `https://tools.ietf.org/html/draft-dempsky-dnscurve-01`, February 2010. `https://tools.ietf.org/html/draft-dempsky-dnscurve-01`.

[18] Alexandre Denis. A high performance superpipeline protocol for InfiniBand. In *Proceedings of the 2011 International Conference on Parallel Processing*, Euro-Par'11, pages 276–287, Berlin, Heidelberg, 2011. Springer-Verlag. `http://dl.acm.org/citation.cfm?id=2033408.2033440`.

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

[20] Todd Fast, Timothy Wall, and Liang Chen. Java native access. `https://github.com/twall/jna#readme`.

[21] Codehaus foundation. Jetty powered. `http://docs.codehaus.org/display/JETTY/Jetty+Powered`.

[22] Codehaus foundation. Jetty webserver. `http://jetty.codehaus.org/jetty/`.

[23] The Apache Software Foundation. org.apache.hadoop.examples.terasort (Hadoop 1.0.2 API). `https://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html`.

[24] The Apache Software Foundation. Welcome to Apache Hadoop. `https://hadoop.apache.org/`.

[25] Sebastien Godard. Sysstat. `http://sebastien.godard.pagesperso-orange.fr/`.

[26] D. Goldenberg, M. Kagan, R. Ravid, and M.S. Tsirkin. Zero copy sockets direct protocol over InfiniBand-preliminary implementation and performance analysis. In *Proceedings of the 13th IEEE Symposium on High Performance Interconnects*, pages 128–137, Palo Alto, CA, August 2005.

[27] William Gropp and Ewing Lusk. Fault tolerance in MPI programs. *Special issue of the Journal High Performance Computing Applications (IJHPCA)*, 18:363–372, 2002.

[28] Paul Grun. *Introduction to InfiniBand for end users*. InfiniBand Trade Association, 2010.

[29] Jeff Hilland, Paul Culley, Jim Pinkerton, and Renato Recio. RDMA protocol verbs specification. `https://tools.ietf.org/html/draft-hilland-rddp-verbs-00`.

[30] Marius Hillenbrand, Viktor Mauch, Jan Stoess, Konrad Miller, and Frank Bellosa. Virtual infiniband clusters for hpc clouds. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, CloudCP '12, pages 9:1–9:6, New York, NY, USA, 2012. ACM. `http://doi.acm.org/10.1145/2168697.2168706`.

[31] David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. Technical Report GIT-CERCS-09-13, CERCS, Georgia Institute of Technology, 2009.

[32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the USENIX 2010 Annual Technical Conference*, USENIXATC'10, pages 11–24, Berkeley, CA, USA, 2010. USENIX Association. `http://dl.acm.org/citation.cfm?id=1855840.1855851`.

[33] InfiniBand Trade Association. *InfiniBand architecture specification*, 2007.

[34] UNH InterOperability Lab. iWARP consortium FAQ. `http://www.iol.unh.edu/services/testing/iwarp/faq.php#What%20is%20iWARP?`

[35] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, pages 743–752, Taipei, Taiwan, September 2011.

[36] IBM journal of Research and Development staff. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2):199–220, January 2008. `http://dl.acm.org/citation.cfm?id=1375990.1376008`.

[37] M. Frans Kaashoek, Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. FLIP: an internetwork protocol for supporting distributed systems. *ACM Transactions on Computer Systems*, 11:73–106, February 1993. `http://doi.acm.org/10.1145/151250.151253`.

[38] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *Analytics Press*, July 2010. `http://www.analyticspress.com/datacenters.html`.

[39] Orran Krieger, Marc A. Auslander, Bryan S. Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria A. Butrico, Mark F. Mergen, Amos Waterland, and Volkmar Uhlig. K42: building a complete operating system. In *Proceedings of the 1st ACM SIGOPS EuroSys conference*, pages 133–145, 2006.

[40] Argonne National Laboratory. The message passing interface (MPI) standard. `http://www.mcs.anl.gov/research/projects/mpi/`.

[41] Matt Liebowitz. Cybercrime blotter: High-profile hacks of 2011. `http://www.securitynewsdaily.com/455-websites-hacked-government-commercial-cybercrime-2011.html`.

[42] J. Liedtke, N. Islam, and T. Jaeger. Preventing denial-of-service attacks on a μ-kernel for webOSes. In *Proceedings of 6th Workshop on Hot Topics in Operating Systems*, pages 73–79, May 1997.

[43] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM. `http://doi.acm.org/10.1145/168619.168633`.

[44] Linux Programmer's Manual. pthread_sigqueue manual page. `http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_sigqueue.3.html`.

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

[46] Viktor Mauch, Marcel Kunze, and Marius Hillenbrand. High performance cloud computing. *Future Generation Computer Systems*, March 2012. `http://www.sciencedirect.com/science/article/`

pii/S0167739X12000647. Available online at http://dx.doi.org/10.1016/j.future.2012.03.011.

[47] Pauline Middelink. Bigphysarea. http://www.polyware.nl/~middelink/En/hob-v4l.html.

[48] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the memory registration process in the Mellanox InfiniBand software stack. In *Proceedings of the 2006 International Conference on Parallel Processing*, Euro-Par'06, pages 124–133, Berlin, Heidelberg, 2006. Springer-Verlag. http://dx.doi.org/10.1007/11823285_13.

[49] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. http://www.ietf.org/rfc/rfc1034.txt. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[50] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. http://www.ietf.org/rfc/rfc1035.txt. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

[51] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23:44–53, May 1990. http://dx.doi.org/10.1109/2.53354.

[52] Raphael Neider. Microkernel construction lecture notes (chapter 4: Tcbs). http://i30www.ira.uka.de/~neider/edu/mkc/mkc.chunked/ch04.html.

[53] Ranjit Noronha, Lei Chai, Thomas Talpey, and Dhabaleswar K. Panda. Designing NFS with RDMA for security, performance and scalability. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, pages 49–56, Washington, DC, USA, 2007. IEEE Computer Society. http://dx.doi.org/10.1109/ICPP.2007.34.

[54] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Parallel Distrib. Technol.*, 5(2):60–73, April 1997. http://dl.acm.org/citation.cfm?id=613783.614014.

[55] J. Pinkerton and E. Deleganes. Direct data placement protocol (DDP) / remote direct memory access protocol (RDMAP) security. RFC 5042 (Proposed Standard), October 2007. http://www.ietf.org/rfc/rfc5042.txt.

[56] B.P. Rimal, Eunmi Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *Fifth International Joint Conference on INC, IMS and IDC*, pages 44–51, August 2009.

[57] Salvatore Sanfilippo. Redis. http://redis.io/.

[58] S. Smith, A. Madhavapeddy, C. Smowton, M. Schwarzkopf, R. Mortier, R.M. Watson, and S. Hand. The case for reconfigurable I/O channels. To appear in Proceedings of the 2012 Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE '12).

[59] Matt Stansberry. 2011 data center industry survey. *Uptime Institute*, June 2011. http://www.uptimeinstitute.com/publications.

[60] Jan Stoess, Jonathan Appavoo, Udo Steinberg, Amos Waterland, Volkmar Uhlig, and Jens Kehne. A light-weight virtual machine monitor for Blue Gene/P. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, pages 3–10, Tucson, Arizona, July 2011.

[61] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM. http://doi.acm.org/10.1145/383059.383071.

[62] Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. Java fast sockets: Enabling high-speed java communications on high performance clusters. *Comput. Commun.*, 31:4049–4059, November 2008. http://dl.acm.org/citation.cfm?id=1456731.1457122.

[63] Zouheir Trabelsi and Wassim El-Hajj. ARP spoofing: a comparative study for education purposes. In *Proceedings of the 2009 Information Security Curriculum Development Conference*, InfoSecCD '09, pages 60–66, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1940976.1940989.

[64] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM. `http://doi.acm.org/10.1145/139669.140382`.

[65] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of the 2010 International Conference on Supercomputing*, pages 57–66, Seattle, WA, USA, November 2011.

[66] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[67] Alan Zeichick. How facebook works. *Technology Review*, July 2008.