Low-Latency Implementation of Network Sockets over Remote DMA

Dimitrios Poulios

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Manolis G.H. Katevenis*

This work was performed in the *Computer Architecture and VLSI Systems (CARV) Laboratory* of the *Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH)*, and was financially supported by a FORTH-ICS scholarship, including funding by the European Union 7th Framework Programme under the **EuroServer** (FP7-ICT-610456) project.

University of Crete
Computer Science Department

Low-Latency Implementation of Network Sockets over Remote DMA

Thesis submitted by
Dimitrios Poulios
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Dimitrios Poulios

Committee approvals: _____
Manolis G.H. Katevenis
Professor, Thesis Supervisor

_____
Angelos Bilas
Professor, Committee Member

_____
Polyvios Pratikakis
Assistant Researcher, ICS-FORTH, Committee Member

Departmental approval: _____
Antonis A. Argyros
Professor, Director of Graduate Studies

Heraklion, March 2015

Abstract

In recent years, changes in the server market have brought power and space efficient server designs, like the Microserver. Such designs utilize large numbers of lightweight compute nodes bundled together to serve scale-out data center workloads. Unfortunately, scalability can often be limited by the quality of internal communication among running nodes, where low throughput and, even more critically, high latency can lead to poor performance. In this work, we explore the efficiency of a Remote Direct Memory Access (RDMA) capable internal network in a Microserver environment.

Applications commonly use the standard Socket API for interprocess communication across networks. Therefore, to take advantage of the aforementioned internal network without modifying existing applications, socket-related system calls have to be intercepted. We implement system call interception in user space, using a modified Standard C Library, in order to bypass the kernel TCP / IP stack. A kernel driver has also been developed to securely perform data transfers via RDMA operations, which require physical addresses. Remote completion notifications of RDMA operations are triggered by a custom hardware Mailbox Mechanism, which also handles communication among nodes, necessary to initiate and close local connections.

By combining these user and kernel space elements, we direct local TCP traffic through our internal network. Evaluation results show a 3x to 5x improvement to the latency, using our system compared to a typical ethernet configuration.

Περίληψη

Τα τελευταία χρόνια, οι αλλαγές στην αγορά των servers έχουν φέρει στο προσκήνιο νέες υλοποιήσεις, όπως ο Microserver, οι οποίες στοχεύουν σε μειωμένη κατανάλωση ενέργειας και οικονομία χώρου. Τέτοιες υλοποιήσεις χρησιμοποιούν μεγάλο πλήθος όχι ιδιαίτερα ισχυρών υπολογιστικών κόμβων, ομαδοποιημένων ώστε να εξυπηρετούν κλιμακώσιμες εφαρμογές προορισμένες για Data Centers. Δυστυχώς όμως, πολλές φορές αυτή η κλιμάκωση περιορίζεται από την ποιότητα της εσωτερικής επικοινωνίας μεταξύ των κόμβων, όπου η χαμηλή παροχή (throughput) και, ακόμα χειρότερα, η μεγάλη καθυστέρηση (latency), μπορεί να οδηγήσει σε κακή απόδοση. Σε αυτήν τη δουλειά, εξερευνούμε την επίδραση που μπορεί να έχει σε ένα περιβάλλον Microserver, η ύπαρξη ενός εσωτερικού δικτύου το οποίο έχει τη δυνατότητα να εκτελεί μεταφορές δεδομένων με πράξεις απομακρυσμένου DMA (RDMA).

Κατά κύριο λόγο, οι εφαρμογές χρησιμοποιούν το Socket API για επικοινωνήσουν μεταξύ τους μέσω δικτύων. Συνεπώς, για να μπορέσουμε να εκμεταλλευτούμε το προαναφερθέν εσωτερικό δίκτυο χωρίς να χρειαστεί να τροποποιήσουμε τις υπάρχουσες εφαρμογές, οι κλήσεις συστήματος (system calls) σχετικές με τα Sockets πρέπει να αναχαιτιστούν (intercepted). Πραγματοποιούμε την αναχαίτιση αυτή στο επίπεδο του χρήστη (user space), χρησιμοποιώντας μια τροποποιημένη έκδοση της Standard C Library, με σκοπό να παρακάμψουμε την επιβάρυνση του πρωτοκόλλου TCP / IP. Επιπλέον, υλοποιήσαμε έναν driver στον πυρήνα, ο οποίος πραγματοποιεί ασφαλείς μεταφορές δεδομένων μέσω πράξεων RDMA, οι οποίες χρειάζονται φυσικές διευθύνσεις. Η απομακρυσμένη ειδοποίηση της ολοκλήρωσης τέτοιων μεταφορών γίνεται με τη βοήθεια ενός μηχανισμού Mailbox, ο οποίος χρησιμοποιείται επίσης για την επικοινωνία που χρειάζονται οι κόμβοι ώστε να δημιουργήσουν ή να τελειώσουν τοπικές συνδέσεις.

Συνδυάζοντας τα παραπάνω στοιχεία, είτε στο επίπεδο του χρήστη ή του πυρήνα, κατευθύνουμε τις εφαρμογές να χρησιμοποιούν το εσωτερικό δίκτυο για τοπικές συνδέσεις TCP. Η αξιολόγηση του συστήματός μας, σε σχέση με μια τυπική διάταξη ethernet, έδειξε βελτίωση από 3 μέχρι 5 φορές στο χρόνο καθυστέρησης.

Acknowledgements

First of all, I would like to thank the advisor of this thesis, Professor Manolis G.H. Katevenis and the rest of the commitee, Professor Angelos Bilas and Dr. Polyvios Pratikakis.

I would also wish to express my sincere thanks to Dr. Manolis Marazakis for his invaluable help throughout this work.

Moreover, I must give my special thanks to the rest of the Euroserver team: Giorgos Kalokairinos, Kostas Harteros, Michalis Ligerakis, Iakovos Mavroidis, Nikos Chrysos, John Velegrakis, Antonis Psathakis and Evangelos Vasilakis.

Finally, I could not forget to express my gratitude to my parents and Nikoletta for their precious support.

This work was performed in the *Computer Architecture and VLSI Systems (CARV) Laboratory* of the *Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH)*, and was financially supported by a FORTH-ICS scholarship, including funding by the European Union 7th Framework Programme under the **EuroServer** (FP7-ICT-610456) project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Microservers

Recent changes in the server market have brought *Microservers* in the spotlight [1, 2]. The continuously rising demand for *energy efficiency*, as well as *space efficiency*, have resulted in the emergence of this new server architecture characterized primarily by its high density of lightweight cores. Microservers typically use processors that are not usually associated with servers, but can be found in mobile devices and have low power consumption.

However, by no means has this new, low-cost architecture come to displace conventional servers. Microservers are designed to undertake specific workloads that can easily execute in parallel, using large numbers of nodes (scale-out workloads) and demand relatively low processing power. Popular examples are web serving applications or data analytics workloads like those using the MapReduce scheme.

1.2 Low latency internal communication

In this work we deal with the internal communication of Microserver nodes. Workloads like the ones mentioned above, even though they are not always demanding in terms of processing power, often perform a great deal of communication among the running nodes and their scalability can be compromised by a slow *internal network*. Low throughput and, even more critically, high latency can lead to underutilization of the cores [18]. Solutions for achieving low latency, like TCP Offloading in the Network Interface (NIC), are usually too expensive to be employed in these low-cost systems.

Internal traffic is typically seen by the operating system of each node as normal network traffic simply heading to nearby destinations; most of the time, the OS is not even aware of their vicinity. Interconnecting server nodes using networks originally designed with larger area specifications is probably an overkill. This statement applies not only to the hardware side of the system, but to the software side, as well. Network protocols like the commonly used TCP/IP are supposed

to handle Wide-Area-Network connections, thus having features not needed for small-scale environments.

Nevertheless, optimizing a Microserver internal network is not a simple task. Utilizing existing high-speed interconnection technologies can be tricky and expensive. Simpler custom hardware solutions can be designed, but this requires effort and extra software support in the OS. Except for this and even more challenging than this, is that in order to reduce the software induced overhead, the whole programming model of the applications may have to be changed; this basically means rewriting them.

1.3 Sockets Over RDMA

Our goal is to allow *unmodified applications* to efficiently utilize an RDMA-capable internal network in order to achieve *low-latency and high-throughput communication*. Our implementation consists of two separate parts executing in user and kernel space.

- In user space, we intercept system calls related to the popular Berkeley Sockets API, to bypass the kernel TCP / IP stack and avoid its overhead.
- In kernel space, we handle data transfers by means of high-speed *Remote Direct Memory Access (RDMA)* transactions, using a custom RDMA driver.

So far in this effort, we support only `stream sockets` (i.e. TCP connection-based streams) and not `datagram sockets` (i.e. UDP connectionless transfers).

1.4 The Euroserver Project

The Euroserver Project[1] aims to explore ARM based Microservers with efficient hardware and systems software support, in order to deliver low-power servers on single interposer chips. Compute Nodes or Compute Units (CUs) are grouped together in a hierarchical network structure, consisting of several *Coherence Islands* (i.e. collections of coherent CUs).

This work was performed as part of the Euroserver Project, using Linux-based systems. The custom RDMA-capable internal network has been utilized in order to enable efficient interprocess communication between the Microserver compute nodes. Testing and evaluation was carried out on the first generation of the Euroserver Discrete Prototype.

1.5 Contributions

The contributions of this work are:

[1] http://www.euroserver-project.eu/

- Design of protocol to run Sockets over RDMA, using per-connection transfer buffers and a Mailbox mechanism to send notifications.
- Interception of system calls in user space, within the Standard C Library, to allow unmodified applications to transparently utilize our system.
- Bypass of kernel TCP / IP stack and implementation of lightweight RDMA-based network sockets.
- Analysis of overheads and a detailed breakdown of latency when using RDMA-based sockets.

1.6 Thesis overview

The rest of this thesis is structured as follows: On Chapter 2 we provide a list of past efforts and protocols related to our work, while on Chapter 3 we give a brief overview of the Sockets API, describing the most common system calls and functions of it. Moving on to Chapter 4, we present our testing environment, the Euroserver Discrete Prototype, listing the hardware specifications and custom features. On Chapter 5, we give a detailed description of how the interception of system calls is performed in a modified `Standard C library (libc)` environment. Afterwards, on Chapter 6 we thoroughly present how we delude applications into thinking of using normal TCP connections, while our RDMA capable internal network is used instead. Finally, after evaluating our system and comparing it with the normal TCP / IP network, on Chapter 7, we conclude on Chapter 8.

Chapter 2

Related Work

2.1 System call interception

System call interception (also referred as *system call interposition*) is most often applied for security reasons, for example in systems offering intrusion detection or application sandboxing. Interception is implemented either strictly in kernel or on user level, but there are also hybrid solutions like [19, 20]. Furthermore, some implementations involve modification of object code either dynamically or statically [21, 22].

User-level libraries (like Infiniband's `SDP` that will be mentioned later) usually employ the library preloading method. This involves using the `LD_PRELOAD` environment variable to instruct the linker to use a particular shared library before all others, interposing this way standard functions. This method, although effective, is more of a temporary solution requiring to always set this environment variable for every application. As we will see in Chapter 5, we use a slightly different technique to intercept socket-related system calls.

2.2 User level networking

Several efforts have been made to avoid the overhead imposed by the kernel TCP/IP stack. This overhead is either caused by the effects of system calls and the kernel context switch or by the multiple copies of data that have to be made during the process.

`Fast Sockets` [23] was an attempt to support Sockets API using a lightweight user-level protocol, `Active Messages` [24], over a Myrinet network. This work included strategies of collapsing protocol layers and simple buffer management, making possible to avoid some copy operations. Another similar effort was implemented on the `Shrimp` platform [25], utilizing custom network interfaces to enable user level communication by allowing applications to transfer data directly between virtual addresses over the network. Compared to our work, limitations of the above efforts are the need for relinking applications to external libraries and the partial

support for shared sockets among cloned processes.

In mTCP [26], the whole network stack is implemented in user space, offering packet- and socket-level batching optimizations. Unlike our system though, applications have to be modified to replace socket calls. Similar techniques have existed for many years. [27, 28]. Furthermore, MegaPipe [29] employs a channel-per-core method by partitioning sockets across cores and exchanging I/O requests and event notifications via pipes between the kernel and user space. In FlexSC[30], system call batching and scheduling is followed to avoid cache pollution caused when kernel and user space code runs on different cores.

Another recent popular technique employs high-performance user level network packet processing, as in netmap, ntop, or Intel's DPDK [31, 11, 12]. The thought here is to provide user space applications very fast access to network packets, often employing a *run-to-completion* model, in order to eliminate allocation and copying costs and take advantage of data locality.

In DaRPC [32], one step further is taken, by offering a *remote procedure call (RPC)* framework integrating RPC processing with network processing in user space by using RDMA. By looking these elements as a joint optimization problem, context switches and cache misses are avoided, leading to better parallelism and latency.

In IX [33], many of the above techniques are employed to improve both latency and throughput in a server environment. The novelty in this case lies in the use of lightweight virtualized operating systems running a single network application and having dedicated resources like CPU cores or allocated memory. This way, coherence traffic is avoided, having an significant effect on scalability, High network performance is achieved by taking advantage of a userland network stack and a selective packet batching technique. Normal applications run in user mode, while the rest of the dedicated OS runs in a intermediate CPU protection ring to assure safe execution of malicious or faulty code.

2.3 RDMA-based systems

Commercial technologies

The idea of using Remote DMA (RDMA) to disengage the processor from the network data moving process is not new. From the beginning of the last decade, commercial systems with RDMA capability have emerged, mostly in the area of high-performance computing, however. Nowadays, the most popular technologies that utilize RDMA are Infiniband, iWARP and RoCE.

Infiniband defines a complete open industry network standard, having its own interface adapters, switches and cables [14]. Infiniband has no standard API; the most widely used software stack is that developed by the OpenFabrics Alliance [13], which also publishes APIs for all common RDMA technologies. The verbs layer is the lowest layer of access to Infiniband hardware. This contains RDMA primitives like RDMA read or write, among others and can also be employed from

user space, completely bypassing the kernel. An upper software layer implemented over verbs in the kernel is the `IP-over-Infiniband` (or IPoIB) layer that provides an interface to the IP protocol. For applications using the sockets API, the `Sockets Direct Protocol (SDP)` has been implemented. This supports only TCP connections and can be used with no or little modifications to the original socket applications. The OS TCP stack is bypassed and various RDMA features like zero copy data transfers can be employed. For other sockets types (datagrams, raw, etc.), the normal path through OS network stack must be followed and then IPoIB to access the Infiniband adapters. Compared to our protocol, although SDP can run unmodified applications and bypasses TCP in kernel, it is still built upon the Infiniband stack, which cannot be characterized as lightweight.

The `iWARP` protocol (also called the `Internet Wide-Area RDMA Protocol`) [15] enables RDMA over TCP/IP. The RDMA interface is the common verbs interface and as a result, iWARP can be easily interchanged with Infiniband for Infiniband based applications. Special `RDMA Network Interface Controllers (rNICs)` must be used on both ends to take advantage of RDMA operations, but regular Ethernet routers or switches can be used to transfer the traffic. Typically, the whole TCP/IP stack is offloaded to the rNIC, allowing *Direct Data Placement* to the buffers and preventing multiple copies of data.

The newest of the three RDMA technologies is `RoCE`, an acronym for `RDMA over Converged Ethernet` [16]. RoCE simply replaces the physical and MAC layers of Infiniband with the Ethernet equivalents. Although it can operate in a traditional Ethernet network, to take the full advantage of RoCE, a converged Ethernet network (NICs, switches) should be used. This is Ethernet with DCB (*Data Center Bridging*) support [17], which eliminates losses due to queue overflows, differentiating from the original design of Ethernet as a *best-effort network*. Moreover, as in iWARP, the endpoint adapters should be RDMA aware. Major shortcomings of RoCE are that is not routable and cannot scale efficiently, although the recent version 2 of the protocol is claimed to address these problems.

Research projects

Along with commercial developments, academic research on RDMA and its applications also flourishes in recent years, delivering custom systems or involving the porting of well known applications. Other efforts include important evaluation and analysis of protocols and standards [34, 35].

One major area of research has been the application of RDMA to *Message Passing* systems. In [36], an MPI implementation (`MVAPICH`) for Infiniband was introduced, noting a considerable improvement in performance, while in `OpenMPI` [37], extra attention was paid to scalability issues. Other projects have dealt with distributed file systems over RDMA [38, 39] or `MapReduce` frameworks like the `Apache Hadoop` [40].

Key-Value stores are data center workloads that have also attracted a lot of attention in the RDMA research field. For example, in [41] the authors have ported

the popular, open source `Memcached` system over Infiniband and give a detailed performance comparison of the different Infiniband modes (verbs, SDP) and the original design over Ethernet. In [42], a custom design, called `Pilaf`, uses RDMA only for `get` operations (RDMA reads), while writing new keys (`put` operations) are serviced normally by the server, to avoid synchronization issues. Inconsistent RDMA reads are detected with what the authors call *self-verifying* data structures, incorporating validation checksums. Following a different approach, the `HERD` system [43] does not use RDMA reads at all, but instead it relies on *two-sided* RDMA verbs, where the responder's CPU is involved and allows an incoming send to be initiated.

In Marlin [44], the authors created a custom RDMA over PCIe network, to boost the inter-rack communication performance in a *disaggregated rack architecture*. In `FaRM` [45], a cluster system was implemented that enables main memory sharing among the nodes with the help of a RoCE interconnect, performing an order of magnitude better compared with recent TCP/IP solutions. A similar work, called `soNUMA` [46], deals with distributed memory remote access latencies by integrating a protocol controller into a node's local coherence hierarchy.

Chapter 3

The Socket API

The Berkeley Sockets (also known as BSD Sockets) is the most commonly used *Application Programming interface (API)* for interprocess communication across computer networks. It has become, almost unaltered, part of the POSIX Specification that all Unix and Unix-like systems follow. Typically, it is part of the operating system and the `Standard C Library (libc)` which provide the respective system calls and functions needed.

3.1 Socket types

A socket is an endpoint of an interprocess communication flow. There are several types of sockets (e.g. for local connections or as a kernel user interface); our focus is solely to those using the `internet protocol (IP)` for communication, also called *internet sockets*. Besides communication domain, sockets have different types based on their communication semantics. The most common are: `stream`, `datagram` and `raw` sockets. Raw sockets work directly with IP packets, not having any interaction with the `transport layer` of the network stack. Whereas, datagram sockets refer to `UDP`-based communication, and stream sockets rely on `TCP`-based connections.

The UDP protocol follows a connectionless, unreliable communication scheme. Data are sent to a destination (differentiated by its address and listening port number), without any delivery acknowledgement. On the other hand, TCP sockets provide reliable, connection-based data transfers and are predominantly used in the internet today. In this work we are working with the latter case only, thus we will next give a short reference to the usage of stream sockets. Figure 3.1 illustrates the whole sequence of an example TCP connection. In the following sections, the most common system calls related to each connection step are briefly discussed. The accompanying listings present the system call prototypes. For complete reference, please see [3].

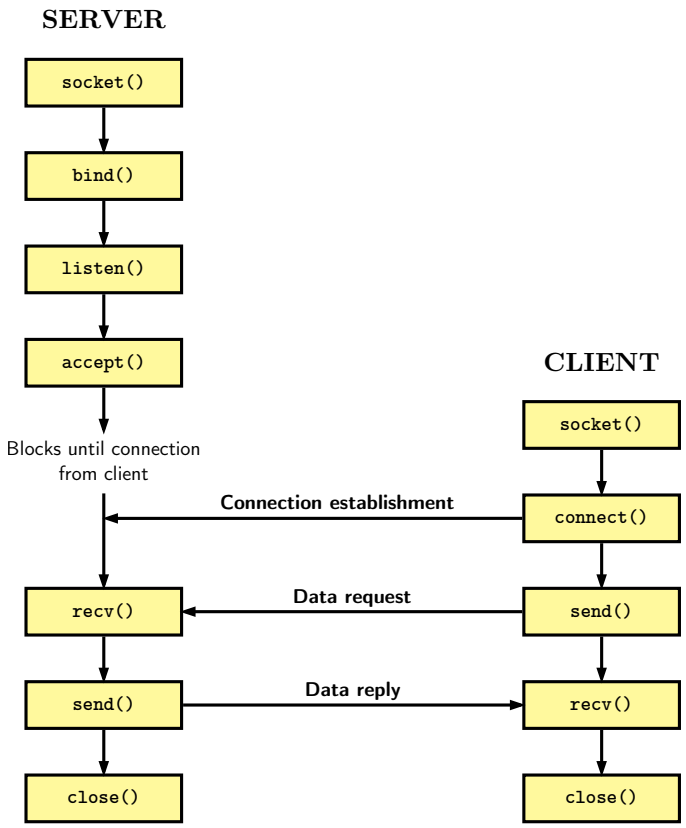

Figure 3.1: TCP connection example

3.2 Server initialization

To create a new socket, both sides have to call the `socket` system call. The `domain` argument must be `AF_INET` or `AF_INET6` to request internet sockets of version 4 or 6 of the Internet Protocol and the `type` argument must be set to `SOCK_STREAM` to indicate usage of the TCP protocol.

Stream sockets require a connection with the two endpoints to be established before data transactions can occur. Therefore, one of the two peers, that is known as the server side, must inform the system that he is waiting for an incoming connection from a remote peer, the client side.

```
int socket(int domain, int type, int protocol);
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen)
int listen(int sockfd, int backlog);
```

Listing 3.1: Pre-connection system calls

In order to hold multiple connections simultaneously, each one is tied to a

different `port` number. This applies to both the server and the client side, though the port number is not the same. The pair of the IP address and the port number is a unique identification key of anyone in the network.

Consequently, the server has to ask first for a particular port number from the OS. This occurs by issuing the `bind` system call. The port number is passed through the `addr` argument. In this structure, it can be optionally requested that the acceptable incoming connections are from a specific network interface of the system.

Afterwards, by calling `listen`, the server is now ready to accept new connections. The `backlog` argument denotes the maximum number of pending connections waiting to be served.

3.3 Connection establishment

As soon as the server starts listening to a port, the client can request a new connection by using the `connect` system call. The address and the port number of the server are given through the `addr` argument. This call blocks until the server responds. A negative return value indicates that the request could not handled, giving some reasoning for this result. For example, the port number could have been wrong.

```
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Listing 3.2: Connection establishment system calls

The server calls `accept` to be notified for each client arrival. When a new connection is accepted, the call returns and the client's details are written in `accept`'s `addr` which is now a return argument. For additional connections, `accept` has to be reissued. One significant difference with the client's connection procedure, is that, for each accepted incoming connection a new socket is created at the server. The file descriptor number of this new socket is the return value of `accept` and this must be used for any subsequent data exchange.

It has to be noted that, it is possible that the `connect` call returns prior to the server calling `accept`. The client can also send data that will be read after the server's `accept`. In other words, the process of accepting a new connection is the kernel's responsibility and the `accept` system call is just the way to inform the interested process.

3.4 Data transfer

Data transfers are bidirectional and occur with the use of `send` and `recv` system calls. Typically, the kernel holds some buffer space to temporarily store the data before the processes use them. When incoming data are already available, a call to `recv` will copy them to the buffer given by the process (`buf` argument). Otherwise, `recv` will block until something arrives. A `len` argument is also passed indicating the maximum number of bytes that can be read – which must not be larger than the size of the process' buffer. The returned value is the number of bytes received and can be less than or equal to `len`.

On the other hand, `send` sends `len` number of bytes stored in the process' `buf` buffer. Again, the successfully written number of bytes is returned. During the call, user's data are only copied to the kernel's send buffer and the actual transfer across the network may happen at a later moment. Consequently, a `send` call will block if the kernel's buffer is full. This event is rare, so `send` usually returns immediately.

```
ssize_t send(int sockfd, const void *buf, size_t len,
             int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Listing 3.3: Basic data transfering system calls

Besides `send` and `recv`, data transactions can be performed with other calls as well. The conventional `write` and `read` act in exactly the same way when dealing with a socket file descriptor. Moreover, several other variations exist. `sendto` and `recvfrom` are ordinarily used with UDP sockets, containing an extra address argument (because UDP sockets don't have a dedicated remote side). For TCP sockets, this argument is ignored. Additionally, the pairs of `sendmsg`/`recvmsg` and `sendmmsg`/`recvmmsg` are similar to the above, differing only in the way (the data structures) data is exchanged with the OS.

3.5 Closing a connection

Closing a socket, ends the connection; any subsequent data transfer requests from the remote side will fail. For this action, there is not a dedicated socket system call, but the common `close` call is used. However, at the server side there are extra sockets for each accepted client, as we have seen. These new sockets must be closed in order to terminate the connections. Conversely, closing the original one results in merely stopping the action of listening to the port. Of course, this can happen before or after the ending of an accepted connection.

The `shutdown` system call allows disabling the connection per-direction – as seen from the caller. For example, a peer can shutdown the reading of a socket, causing this way the remote side's writes to fail, while reading is done normally. A call to `close` still has to be performed though, to release the socket resources.

```
int close(int fd);
int shutdown(int sockfd, int how);
```

Listing 3.4: System calls to terminate connections

3.6 Miscellaneous

Supplementary system calls like `getsockname` or `getpeername` also exist. The first returns the local address a socket is bound to and the second returns the address of the remote peer connected to the socket. Moreover, the pair of `getsockopt`/`setsockopt` is often used. These system calls are responsible for modifying or reading the current values of the numerous *socket options*. Socket options allow tweaking the parameters of a socket and offer more advanced use of the connection. For instance, values like the kernel buffers' sizes, socket priorities, TCP timeouts et al. can be set. One of the most common options is the `SOCK_NONBLOCK` which makes the socket non-blocking. Not having any received data then, makes `recv` return immediately. The `select` system call can be used in this case to notify for any data arrival.

Besides support from kernel, there is a list of auxiliary functions residing in the Standard C Library, or `libc`. For example, the `gethostby*` family (e.g. `gethostbyname(), gethostbyaddr()`), handles the task of converting (by interacting with the DNS service of the OS) the symbolic name of a remote host to a valid IP address. Other useful functions are `htonl, htons, ntohl` and `ntohs` used to convert values like the port numbers from or to *Network Byte Order*, which is always *big-endian*, whereas the host system's endianness depends on the processor's architecture.

3.7 Socket vs RDMA semantics

Using the Sockets API, applications communicate through byte-oriented network streams. On the other hand, RDMA has message-oriented semantics. For example, a TCP application can normally send more bytes than the receiver expects. The surplus will be simply read from the next read call. Whereas with RDMA, fixed sized buffers exist that impose limitations. Only a partial transfer can take place in the previous case and the receiver would have to acknowledge the reception to ask for more. Thus, an application written with stream semantics could result in data loss. Furthermore, another difficult situation is the handling of multiple threads sharing a common socket. A stream connection delivers data successively in this case, whereas with RDMA, receive buffers will have to be protected to ensure proper transfer.

Examples like these bring out the semantic differences of these two data transferring paradigms. Consequently, to port applications from one to the other, so-

lutions involving data buffering or additional communication/negotiation between the two endpoints would have to be followed. This way, data integrity can be assured, perhaps by compromising though, part of the system performance.

Chapter 4

The Euroserver Discrete Prototype

This work has been conducted on Version 1 of the discrete prototype implemented for the Euroserver Project. The prototype consists of two Avnet ZedBoard development boards[1], equipped with Xilinx Zynq-7000 all programmable System-On-Chip. The latter includes an ARM Cortex A9 2-core processor coupled with Xilinx 7-Series FPGA programmable logic. Moreover, the board features a 512 MB DDR3 main memory, IO pins for external connectivity (through an FMC Connector) and supports SD Cards up to 4 GB.

4.1 Custom interconnect

The two boards are connected together with a FMC-to-FMC cable, as can be seen in Figure 4.1. 15 LVDS pairs per direction are used connecting the processing system of each side through the AXI Bus. The bridging of the two AXI buses is implemented by the Xilinx AXI2AXI IP cores.

The processing system clock runs at 667 MHz, the AXI interconnect runs at 100 MHz and the board-to-board connection at 300 MHz DDR. The maximum throughput per direction between the two boards is 9 Gbps.

4.2 Address translation

Except for the interconnection between the nodes, several sharing features have also been added to the prototype. Remote read and writes can be performed from each side and functions like remote swap or remote page borrowing are also implemented. Coherent remote accesses are performed through the ACP Port, which snoops on the memory bus. Alternatively, when no coherency is needed, the HP Port can be used, eliminating the extra overhead.

[1] http://zedboard.org/product/zedboard

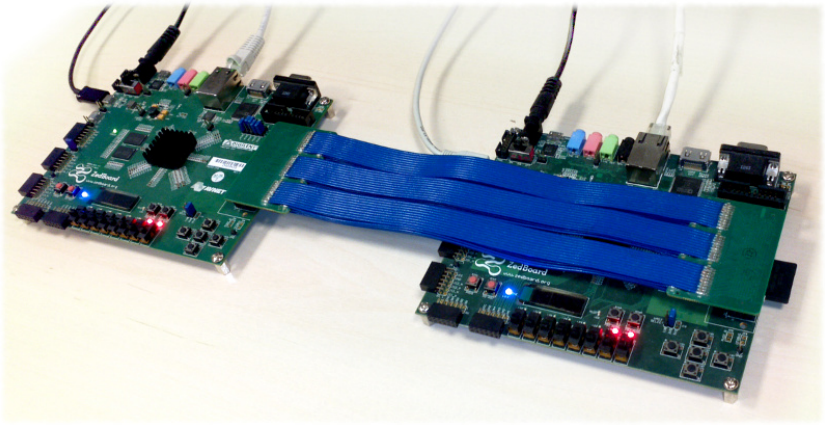

Figure 4.1: The Euroserver Discrete Prototype, version 1

In order to make remote accesses feasible, an *Address Translation Mechanism* has been created using custom FPGA logic. A board "sees" its local 512 MB of physical memory start normally at address 0×00 00 00 00. In contrast, remote memory can be accessed in two ways, either using the coherent `ACP Port` or the `HP Port`. With the first option, the whole remote physical memory can be accessed, as opposed to the `HP Port` where only the second half of the remote memory is seen.

To use `ACP`, local addresses that for board #0 start at 0×00 00 00 00, can be seen by board #1 start at its 0×40 00 00 00. We have the exact same behaviour in the opposite way. Contrariwise, for `HP` accesses, only 256 MB are mapped: Local 0×10 00 00 00 is for the adjacent board its 0×60 00 00 00. Table 4.1 summarizes remote address mappings.

From the above, it can be seen that the transformation of addresses to remote addresses is a simple procedure. One has to take the desired address and perform a `logical OR` with either 0×40 00 00 00 for `ACP`, or 0×60 00 00 00 for `HP`. The second though, is only valid for addresses of the higher 256 MB.

4.3 DMA engine

The prototype has been equipped with two `Direct Memory Access (DMA)` engines, the `AXI DMA` and the `AXI Central DMA (CDMA)`. In conjunction with the address translation feature, `Remote DMA (RDMA)` operations are feasible. For the purposes

Type	Size	Range (Local)	Remote mapping starting address
Local	512 MB	0×00 00 00 00 – 0×1F FF FF FF	
Remote ACP	512 MB	0×40 00 00 00 – 0×5F FF FF FF	0×00 00 00 00
Remote HP	256 MB	0×60 00 00 00 – 0×6F FF FF FF	0×10 00 00 00

Table 4.1: Local / Remote address ranges and mappings

Offset	Name	Description
0×00	CDMACR	CDMA Control
0×04	CDMASR	CDMA Status
0×08	CURDESC_PNTR	Current Descriptor Pointer
0×0C	Reserved	N/A
0×10	TAILDESC_PNTR	Tail Descriptor Pointer
0×14	Reserved	N/A
0×18	SA	Source Address
0×1C	Reserved	N/A
0×20	DA	Destination Address
0×24	Reserved	N/A
0×28	BTT	Bytes to Transfer

Table 4.2: CDMA register space

of this work, we only make use of CDMA, that can perform *memory-to-memory* transfers, without imposing any alignment restrictions. The engine operates in two different modes: the *simple* and the *scatter gather mode*. [4]

The register space of CDMA is depicted in Table 4.2. All registers hold a 32 bit value and the offset column shows the distance (in hexadecimal) of each register from the engine's base address in the system. In Control Register (CDMACR), the mode of operation is set and other parameters like the interrupt creation are also controlled. The current state of the engine (being idle, interrupts produced et al.) can be obtained from Status Register (CDMASR). For producing a Simple Mode transfer, the source address, the destination address and finally the number of bytes to be sent, have to be set in registers SA, DA and BTT respectively. Storing a value in BTT immediately fires the DMA operation.

The Scatter Gather mode – which we exclusively use – offers the ability of multiple consecutive DMA transfers from and to different memory locations. Every individual transfer is controlled by a separate DMA Descriptor, which is a structure stored in memory (for RDMAs, in the sender's memory), besides the actual data. As a result, before any scatter gather RDMA operation, one or more descriptors have to be prepared. Again, these contain 32 bit values, in the form shown in Table 4.3. Like before, the source and destination have to be written (SA and DA registers), and the number of bytes in CONTROL. The STATUS field must hold a zero value, so that later the engine writes there the outcome of this transfer. Finally, NXTDESC_PNTR is used to point to the starting address of the next descriptor to be used, creating this way, a chain of descriptors.

Following the descriptors' preparation, we can initiate a scatter gather operation. The current descriptor pointer register (CURDESC_PNTR) must be given the starting descriptor's address and, thereafter, the tail descriptor pointer (TAILDESC_PNTR) must be written to let the engine know where to stop. This initiates the transfer. Later on, when the descriptor shown by the tail pointer is reached, the

Offset	Name	Description
0×00	NXTDESC_PNTR	Next Descriptor Pointer
0×04	Reserved	N/A
0×08	SA	Source Address
0×0C	Reserved	N/A
0×10	DA	Destination Address
0×14	Reserved	N/A
0×18	CONTROL	Transfer Control
0×1C	STATUS	Transfer Status

Table 4.3: CDMA scatter gather descriptor format

engine processes it and then pauses. If there are more descriptors left in the chain, a new write of TAILDESC_PNTR restarts CDMA.

One significant detail regarding the CDMA is that all address values must be physical addresses. This fact complicates using the engine from user space. Moreover, it has to be noted that, though a scatter gather descriptor is 32 bytes long, each new descriptor must be aligned to a 64-byte address; this has nothing to do with the data alignment – as we mentioned, for this exist no restrictions.

4.4 Mailbox mechanism

A custom Mailbox Mechanism has been implemented, allowing to easily produce interrupts across the system. It contains a FIFO which can hold up to 1024 64-bit words. Several types of interrupts are supported: when an enqueue has happened, when the FIFO becomes non-empty from an empty state or vice versa and when the queue has become full. Enabling individual interrupt types is done via the control / status register.

The basic Mailbox operation is performed by reading or writing in two different memory locations belonging to the device. The first one conducts a blocking read and a simple enqueue when writing. In presence of data, a simple dequeue occurs, whereas an empty FIFO causes the read to block. The second memory location, on the other hand, does not block on read and when writing, it produces an interrupt apart from the enqueue (as long as the respective interrupt type is enabled, of course). A non-blocking read from an empty mailbox returns the fixed value 0×CAFE BEBE DEAD BEEF. Table 4.4 summarizes these different behaviours.

One of primary functionalities of the Mailbox in our work, is the ability to trigger *Remote Interrupts*. Of course, this device is not part of the physical memory, thus it is not related to what we have previously mentioned about remote memory mappings. Its actual location in the local address range is 0×80 00 10 00. However, in a similar way with the above, an extra remote mapping has also been created. This way a board is able to access the remote mailbox by using the base address 0×70 00 10 00. As we will see on Chapter 6, we can even combine the CDMA engine

Offset	**Register Type**	
0×08	Control / Status Register	
Offset	**Read Action**	**Write Action**
0×00	blocking dequeue	enqueue
0×10	non-blocking dequeue	enqueue + interrupt

Table 4.4: Mailbox registers

with the `Mailbox`, so that interrupts are produced as part of an RDMA operation. This takes place by using as destination address, the address of the `Mailbox` and setting the number of bytes to 8.

Chapter 5

System Call Interception

Applications use the network subsystem of the Operating System through the system calls of the `Socket API` described in Chapter 3. Since we need to leave applications *unmodified*, any intervention has to be made after these system calls have been issued.

5.1 User space interception

One potential solution is to modify the running `Linux Kernel`. This is *kernel space interception*. Considering the fact that we are going to use a custom communication scheme, any intervention has to happen as soon as possible, close to the system calls' entry points in the kernel. Thereby, the latency added by the heavy `TCP/IP stack` can be avoided.

However, another significant source of overhead is caused by simply entering and leaving the kernel space. The processor has to switch mode and the total delay can be thousands of cycles. In order to avoid this overhead, as well, *user space interception* has to be performed. Unfortunately, the use of the DMA engine imposes some restrictions. The device could in fact be controlled directly by a process from user space, but this does not allow efficient arbitration among many users. Second, and more important, we cannot allow exposing physical addresses, that the engine needs to operate, to the user space.

From the above it seems inevitable, that even if we intercept execution from user space, eventually we will not able to avoid the kernel completely. However, we persist following this path mainly for two reasons:

1. To explore the potential benefits of avoiding the kernel, even some times. For example, a `send` call may not be able to send data if the remote side is not ready for reception.

2. The potential future availability of new sophisticated DMA engines working with virtual addresses. This is one of the prospective goals of the `Euroserver Project` for the foreseeable future.

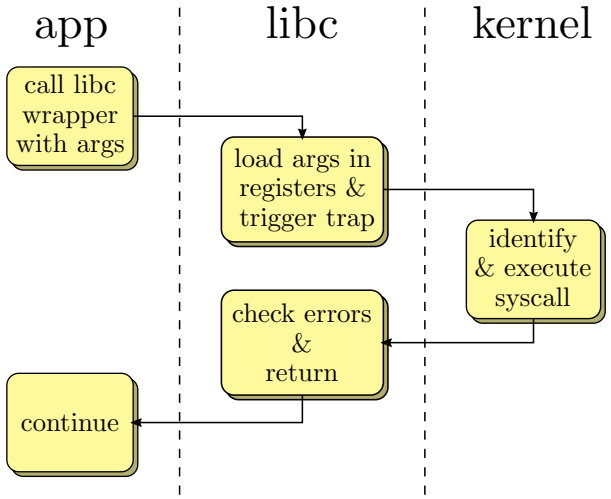

Figure 5.1: System call execution flow

5.1.1 System call wrapper functions

But, how can a system call be intercepted in user space, after the application has issued it? The answer here is that the application has not made the system call by itself.

Making a system call, means triggering a software interrupt (or a software trap) to force the processor to enter kernel space. This procedure demands low level, assembly programming and thus, is different for every architecture. For this reason, system calls are almost never handled by the programmer. In Unix & Unix-like systems this is responsibility of the `Standard C Library`, also known as `libc`. As a matter of fact, applications always call simple `libc` functions, that have the same name as the original system call and are called *System Call Wrappers*.

Initially, a system call wrapper prepares the call. The type of the call and the arguments it needs, have to be placed at specific locations which are usually processor registers. This is where the kernel will look for them. Afterwards, it can trigger the trap. This is illustrated in Figure 5.1. When the kernel returns, in case of an error, the result is handled according to the `errno` interface of the `C Language Standard`. Eventually, execution returns to the normal flow of the user application.

What we do in this work is intercepting the system calls by injecting code within the system call wrappers. There is hardly any software not using the libc functions to interact with the kernel. This is true, even when using other languages like Java or OCaml or scripting languages like Python, Perl, Ruby et al.. These languages usually define their own socket interface, either built-in or as an extension library, but ultimately, all end up using `libc`. One case, though, that we obviously cannot deal with, is any statically built binary.

Rather than modifying `libc` as a whole, there is the alternative method of

LD_PRELOAD method. LD_PRELOAD is a parameter of the dynamic linker that is given as environmental variable and allows using a custom library as an interposer before other linked libraries. Typically, the first library to be examined for symbols is libc, so by placing the custom library before, even system call wrappers can be overridden. However, employing the preload method is more of a temporary solution not suitable for normal use, so we have chosen not to follow it.

5.2 The GNU C Library

The most commonly used libc version in Linux world today is the GNU C Library (glibc) created by the GNU Project. In our prototype we use the version 2.15 of glibc. On this version, ARM support is not included in the normal distribution of the library, but it needs to be downloaded separately as an add-on. In newer versions though, ARM support has been fully incorporated in glibc.

5.2.1 System call wrappers implementation

glibc features a very complex building environment due to the fact that it supports several different architectures, kernels and unix specifications. The directory named socket, for instance, contains most of the socket system call wrappers but these are just *stub* versions of the real functions; they always return an error. This kind of code is included in the final library, only when something is not supported in the target architecture.

The real implementation of system call wrapper functions is usually found under the sysdeps direcories which contain system dependent code. Generally, they can be divided in three different types: assembly, macro, and bespoke [5].

Assembly wrappers

Most of the system calls are handled by assembly wrappers. These simply perform the basics, as described in 5.1.1. The only things that differ from one another is the system call number and the number and type of arguments it has. For this reason, there are not source files for every specific system call, but the code is generated at build time, using assembly templates. The use of assembly eliminates any unnecessary overhead caused by a compiler.

The sh script in sysdeps/unix/make-syscalls.sh is used to parse the various syscalls.list files and export necessary information. These files exist in many locations inside the sysdeps folders and have a special format, with each line representing one call and its attributes. When the library is being built, several syscalls.list files, either generic or architecture specific, are read and this way the whole list of assembly system call wrappers is determined. For each one, the above script compiles the file sysdeps/unix/syscall-template.S having passed at the same time suitable variables and values to the preprocessor. The macros in

this file produce the final assembly code with the help of other macros defined in the various `sysdep.h` files.

Macro wrappers

Some system calls require more work to be done before or after. For example, in some system calls, a little different interface is exposed by `libc` than the one the kernel actually uses. Such cases are handled by the so-called macro wrappers. Their code is defined in C language files and the actual call is made by inline macros again defined in a `sysdep.h` file.

For example, the file `sysdeps/unix/sysv/linux/sendmmsg.c` contains the Linux implementation of `sendmmsg`. This wrapper will be different for any architecture because it uses the `INLINE_SYSCALL()` macro which includes inline assembly code.

Bespoke wrappers

There are a few system call wrappers that do not use the standard assembly or C inline macros. In the future, probably these will be changed, too. No socket-related calls belong to this category.

5.2.2 `libpthread` integration

The `Native POSIX Thread Library (nptl)` is a linux implementation of the `POSIX Threads (pthreads)` api that is now integrated in `glibc`. Besides the standard `libc` library, libpthread is also produced to be used by multithreaded applications.

`libpthread` redefines several system calls and because of this, it has its own `sysdeps` directory structure (`nptl/sysdeps`). The `sysdep.h` files we saw earlier are now called `sysdep-cancel.h`. This is because these system calls act as a pthread cancellation point when used in `libpthread` – they are checking if their thread has been cancelled. Thus, their implementation is a little different than the normal one.

An interesting thing is that, although two distinct libraries exist, `libc` also includes some functions that normally belong to `libpthread`. Besides this, even some system call wrappers inside libc use `nptl`'s versions – those created by the macros of `sysdep-cancel.h` files.

5.3 Our interception method

In order to intercept the system calls inside `glibc`, we must inject our code within the wrappers. This is quite easy for macro wrappers, since they are written in C. For assembly wrappers, we will have to call our functions from the assembly code. This is more complicated and we will discuss the details right after.

Because we need unmodified applications, our functions cannot form a new library, as nothing would be linked with it. Another solution could be to make `libc` linked with our library, but this would demand big changes. Therefore, we integrate our code inside `glibc`.

In the root folder of the library we create a new subfolder called `euroserver`. In `ports/sysdeps/unix/sysv/linux/arm/Subdirs` file then, we add this name, so that the build system becomes aware of it. A `Makefile` will now be searched out in our folder. This must have a necessary format in order to be valid. At least, the `Rules` file from the root folder must be included, and the variables `subdir` and `routines` must be defined. The first just contains the name of our folder and the second all the files that need to be compiled, without their name extension. Other variables could be used, like `headers` which would make our header files to be included in the final installation of the library. Since we need our `euroserver.h` file only for building the modified `libc` and since we want it to be visible from everywhere (we call our functions from other folders, e.g. from `sysdeps`), we just place it inside the basic `include` folder of the root directory, without setting the `headers` variable.

There are many special `glibc` identifiers to control the visibility of new symbols in the code. Without setting anything, everything remains internal to the library and this is the behaviour we need: All our functions and global variables (some exceptions will be discussed later) will be visible by all code of `libc`, but invisible to applications or libraries linked to it.

5.3.1 Assembly templates & ARM calling conventions

For the assembly-made wrappers we have modified the `make-syscalls.sh` script to identify one extra custom option. Then, in the `syscalls.list` files, to those system calls we want to intercept, we have added this special option (simply by prepending a single character). As a result, the script builds these system call wrappers using our custom macros defined in `sysdep.h`. These macros are the same as the originals, with the addition of a call to one of our functions in the `euroserver` folder. There is one dedicated function for each intercepted system call.

But, how we can call our functions from the assembly code? Recall that this assembly code represents normal functions (the wrappers), that were called from user code. Thus, according to the *ARM Calling Conventions* [6], the first four arguments are passed to the wrapper through registers `r0` to `r3`. All the rest are put in the stack. This is of course valid only for arguments not bigger than 32 bits[1], but there is not any such socket system call, that would have to be intercepted in our case. On the other hand, the kernel expects the system calls' arguments in registers `r0` to `r6`. The system call number is part of the assembly command.

Our goal is: From assembly, call an external function that will see exactly the

[1] `int`, `long` are 32 bits and `long long` is 64 bits on ARM

same arguments and when it returns everything must be returned to its initial state, like the call never happened. In reality, not everything has to be restored. For example, if the subsequent system call has only two arguments, we must assure that registers `r0` & `r1` are preserved; the wrapper will only use these to make the software interrupt afterwards. Whereas, for a 6-argument call we must make sure that registers `r0` – `r3` are preserved, plus that the stack is in its original state. However, besides the registers that hold the arguments, one additional register that has to be saved at all times is the `Link Register (lr)`. This contains the return address and as a result, when we enter the wrapper, it contains an address from the wrapper's caller. We call our function using the `branch & link (bl)` instruction, which apart from updating the `program counter`, it also stores the return address – an address inside the wrapper – in `lr`. Therefore, `lr` has to be stored every time in order to be able to return to the application.

For system calls with four arguments or less the procedure is simple: We push the `link register` to the stack, and from `r0` to `r3` registers, as many as the arguments. We want our function to have access to all the original arguments and this is indeed true since we have not changed the registers. The reason we push up to four of the first registers though, is that these are *scratch registers*. The callee can change them and therefore they need to be saved by the caller, if necessary. Later on, when the function has returned, we act in the exact reverse way, by popping the same registers from the stack. Everything is then ready for the trap.

On the other hand, when we deal with more than four arguments, a little more work has to be done. As an example, we provide Figure 5.2 which illustrates three different states of the stack for a 5-argument system call. On ARM, a full descending stack is used, so a push operation grows it downwards and the value of the last push is always the current value of `sp`. We denote this value with the bold text. At the initial state (i), the wrapper has just been called, so the first 4 arguments are in the registers and the 5^{th} is in the stack, on (relative) position 0. The `xxx` value on the top means that the data there has nothing to do with us. As before, all 4 scratch registers (`r0 - r3`) must be pushed together with `lr`.

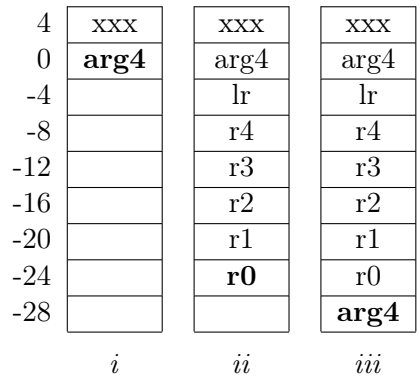

Figure 5.2: Stack operations for a 5-argument system call

However, if we only do this, our function which also expects five arguments, will consider the current stack value as its last argument. As a consequence, the value of `r0` will be passed twice and not the correct one. So, we must also push the value of the 5^{th} argument (`arg4` in the figure). But, before we do this, we have to read its value from the stack; and therefore we need another register. If we use one of the first four, we will destroy the other arguments and if we use, for example, `r4`, it will not be a scratch register. This means that the code which called the wrapper expects `r4` not to have changed by the wrapper. So, at (*ii*), we also push the value of `r4` besides the other registers. Afterwards, we load in `r4` the value `arg4` and then, at (*iii*), we push `r4`. Now, both our called function sees the correct arguments and everything is saved. Finally, to clean up after the function's return, we perform the reverse actions. The only exception here is that we do not need to pop the first value – it is useless, but we can simply move the stack pointer (add the value 4), to avoid the load from memory.

5.3.2 Custom return macros

In the previous sections, we have described how our injected functions get the correct arguments, but we have not mentioned what and how they return to the system call wrappers. In general, we want to inject our code not only to be able to snoop each call; what we actually need is to control it as well. We must have the ability to choose whether the original system call continues like we never existed, or decide to override it (and the kernel) and return to the caller our value.

Preventing the subsequent software trap is easy. When our assembly code restores the values of the registers, it can simply give the instruction `pop {pc}` instead of `pop {lr}`. The latter restores the old value of the link register from the stack. This value is the return address of the wrapper's caller, so if we load it to `pc` instead of `lr`, we will immediately return there. Following this, what the original caller will see as a return value, is the value of `r0` register. This is the convention used on ARM systems.

Eventually, in order to be able to control this behaviour from within our (C code) functions, all of them have a return type of `long long`. This means that they return double word, which spans two registers. The procedure call standard of ARM states that, for these occasions, the return value is written to both `r0` and `r1` (first and second half). The `r1` value is checked by our assembly code, after the function, and if it is -1, the system call is cancelled. In this case, the caller will normally see `r0` as the return value. Otherwise, the call continues and `r0` will be then restored to the value of the first argument.

```
#define EUROSERVER_CONTINUE return (long long)0
#define EUROSERVER_ABORT(ret) \
    return 0xffffffff00000000LL | ret
```

Listing 5.1: Return macros

For our convenience, two macros have been defined to perform the aforementioned. These are shown in Listing 4.1. The first makes the system call continue normally, while the latter aborts it and returns our value.

5.3.3 Pre / Post kernel interception

Until now, we have been able to intercept a system call before it enters the kernel. This allows us to abort it when needed. Nonetheless, there are occasions when what we need is to interpret the results of a system call. This entails running the call normally in the kernel every time and then see what it returns when it is finished. The most common example is the case of `accept`, which we will analyze in Chapter 6.

For post kernel interception, our assembly code injected in the wrappers' building templates is different and custom for every case. Luckily, the cases, as we will see, are not many. Again, we call our functions from there, but this time the arguments passed to them are different. Depending on the call, the kernel's return value is passed along with some of the original arguments. These arguments may be output arguments of the system call, using the *pass by reference* technique.

At the end, the value returned to the caller of the wrapper is the value returned from our function (usually an `int` now). This way we can change it if needed, although we do not in most cases; we simply return the kernel's value as is.

In Figure 5.3, the previous depiction of a system call's execution flow (Figure 5.1) is updated to show these two different ways we intervene in `glibc` to inject our code and make the interception.

5.3.4 `libpthread` & global symbols

To include our modified system call wrappers in `libpthread`, the `sysdep-cancel.h` file had to be modified in a similar way to the normal `sysdep.h` file. Apart from this, our functions in the `euroserver` folder could not be found by the linker (since they are internal to libc) during `libpthread`'s build and in order to overcome this, a `euroserver.c` file was created inside the `nptl` directory, which has simple `include` commands only for the necessary files from the `euroserver` folder. This way, these functions are always the same with the originals. Finally, by adding this new file in `nptl/Makefile`, everything gets built without errors.

The most important problem with `libpthread` though, is that the global variables of our `libc` code are not visible to this library. In this case, we cannot make the above trick, which we did for our functions, because by redefining the global variables, each library will possess its own versions of them. Because our modified system calls can be called from both these libraries, errors from different values will be created. For example, the fact that a socket file descriptor refers to a local connection is kept by a global variable. This will be set from the `connect` system call in the client's side. If the application is linked with libpthread, `connect` from this library will be used. On the other hand, a later use of a system call that is

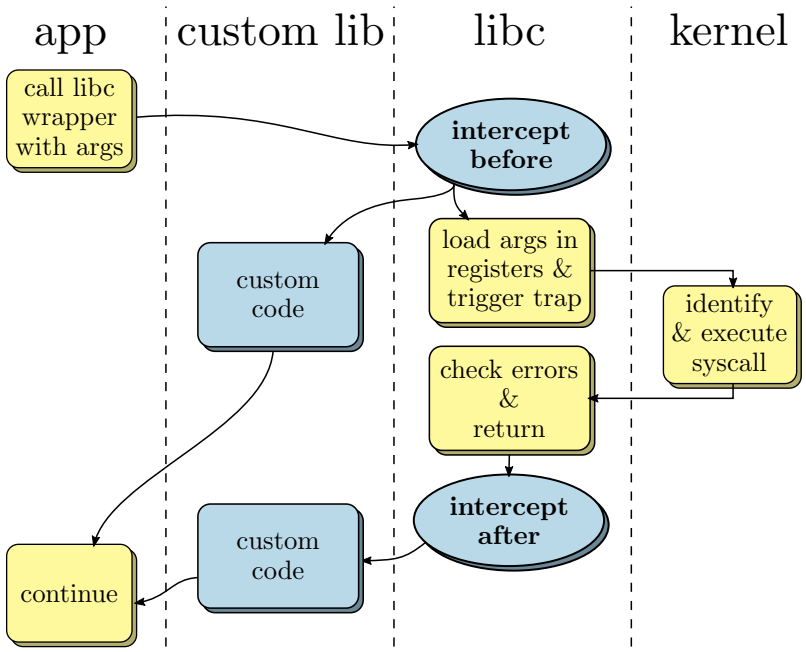

Figure 5.3: Intercepting a system call before or after the kernel

not part of the threads library, will use `libc`'s version of the global variable that has a zero value, which is wrong.

The only way to solve this, is by exporting the symbols, to be visible outside `libc`. This is done by adding their names in the `nptl/Versions` file, as well as by creating our own `euroserver/Versions` file.

5.3.5 The `Run time dynamic linker`

One last difficulty that we encountered with `glibc` was building the `Run time dynamic loader`. This is the loader of all other dynamically linked libraries to a binary. Therefore, it cannot contain functions from other libraries. Unfortunately though, it needs some system calls and those are included in the loader (the wrappers). One of them is `write`, for instance, which is one of our intercepted calls. Thus, our own modified assembly version wrapper was to be included and this caused an error. Hopefully, the build system defines the `IS_IN_rtld` preprocessor variable that is set during the build of the loader. Checking for this value in `sysdeps/unix/syscall-template.S`, ensures that the non-modified wrappers enter the loader.

Chapter 6

Sockets Over RDMA

In this chapter we will present how we create and use TCP connections through our custom RDMA network. To create the delusion of TCP sockets we intercept the socket-related system calls in user space and keep track of particular socket file descriptors that correspond to our local connections. All data transactions are performed with RDMA operations that must be executed from kernel space. For this purpose, a custom RDMA driver has been developed. The user & kernel space cooperation is done by our injected code intercepting system calls in `libc`.

6.1 Kernel driver

A *linux character device*[1] driver has been created to perform all RDMA transfers. There are two main reasons why it is necessary to control RDMA transfers from kernel space:

1. Arbitrating multiple users of the DMA engine
2. Not exposing physical addresses to user space

The driver assumes it is the sole user of the `CDMA` engine and takes full control of it. Moreover, extensive use of the `Mailbox mechanism` is made to send remote interrupts and exchange messages with the remote side.

6.1.1 Driver loading

The driver comes as an external module to the kernel and must be loaded to be used. This procedure is handled by the available *init script*. The init script must be run at privileged (super user) mode.

At first, the virtual file `/proc/devices` is examined to check that the driver is not already loaded and afterwards, the `insmod` command loads the module to the

[1] In Linux, a *character device* is one that can be accessed as a byte stream, like an ordinary file.

kernel. In order to use the driver by user spaces processes, a special device file has to be created. For this reason, `/proc/devices` is scanned again and this time it contains an entry with the name `euroserver` – the name of the driver – and its major device number. This number is passed to the `mknod` command to create the character device file `/dev/euroserver`.

6.1.2 Device initialization

During the module initialization, page table entries for `CDMA` and `Mailbox` are created to provide access to their configuration registers. Then, the devices are reset and configured.

At `CDMA`, regular and error interrupts are enabled and the scatter gather mode is selected. At `Mailbox`, two interrupt types are enabled: the "enqueue" interrupt and the "full FIFO" interrupt. Finally, these interrupts are registered to be handled by the *interrupt handlers* of the driver.

Several types of messages are served by `Mailbox`. Each one of them has a `message ID` and usually a `connection ID`. These are found in the 4 most significant bits and in the next 12 bits of the 64-bit message, respectively. With this configuration we can support a maximum of 16 message types and 4096 simultaneous connections. Currently, only 6 message types are used and the rest are reserved for future use. All different message categories will be presented in detail, moving on in this chapter.

6.1.3 RDMA descriptors

Before the RDMA driver can be used, `RDMA descriptors` have to be created. A set of pre-allocated descriptors is used for all transactions. These are permanently connected together.

In a 4 KB space, 64 descriptors are created. Each one of them has a size of 32 bytes but also requires to be aligned to a 64-byte address. As a result, 32 bytes of padding are added. Every descriptor points to the next via the `NXTDESC_PNTR` field (Section 4.3) and the last one to the first, forming a circular linked list.

The descriptors are protected by a lock. This is a *kernel semaphore* that one must hold to edit any descriptors. As we will see in Section 6.3.4 the number of descriptors for every transaction is not known beforehand. Therefore every sender has to lock the list to avoid overwriting descriptors used by others.

6.1.4 Opening from user space

Processes that create TCP connections within the local network have to use the RDMA driver. Calls to the driver are, most of the times, simple `write`s, passing information to the driver in the buffer argument. For instance, a `"c"` string makes a connect request whereas a `"r14"`[2] means that data must be received from

[2] actually, connection IDs are passed in binary format

connection #14.

Before the driver can be used by a process it must be opened. This occurs the first time a process calls the `socket` system call to create a new internet socket. In the intercepted call, it is checked that the domain argument refers to a network socket and then the driver is opened. The returned file descriptor is stored in a global variable so that every other intercepted system call can use it. In case the driver is not loaded in the system, the normal TCP/IP network path will be followed.

Apart from opening the driver, a special configuration file is also read during the `socket` call. The `/etc/euroserver` file must contain the `IP address` of the remote node. This address is stored in another libc global variable (`peers`) and is checked to determine if a destination is local or not. To support more than one remote nodes, `peers` has the form of a linked list.

6.2 Connection establishment

Before any data transactions can happen, a connection must be established between the server and the client. The client must call `connect`, with the prerequisite that the server is listening to the same port. The accepted connection will then be handed over to the server by using the `accept` system call.

The overview of this procedure regarding our local connections is shown in the timing diagram, in Figure 6.1, where the interactions among the various software elements are emphasized. The thin lines indicate the execution flow, while the thick ones, solid or dashed, show if the code is either running or waiting. Function names are written in blue text[3], whereas black text is additional information. Before explaining in detail the procedure, some preliminary information must be given.

6.2.1 Connection IDs

Each connection must have a unique `connection ID`, in order to be distinguishable among others. We aim to support networks with more than two local nodes and therefore each node could have multiple local open connections. As a result, these connection IDs cannot be global – this would require everybody to get informed about a new connection. Another solution would be to have dedicated connection IDs for each remote peer. This, apart from wasting resources, would require every sent message to include both the connection and the sender id. Depending on how many users and how many connections per remote user we would have to support, this information could be many bits per message.

Instead, we chose to use a global set of connection IDs for each node. When `Peer A` tries to connect with a remote side, it will initially search for the first available of its IDs, let's for example assume this is number 3. Subsequently, it will inform `Peer B` that it will identify this particular connection as connection

[3]functions of real system calls in linux kernel always have names beginning with `sys_`

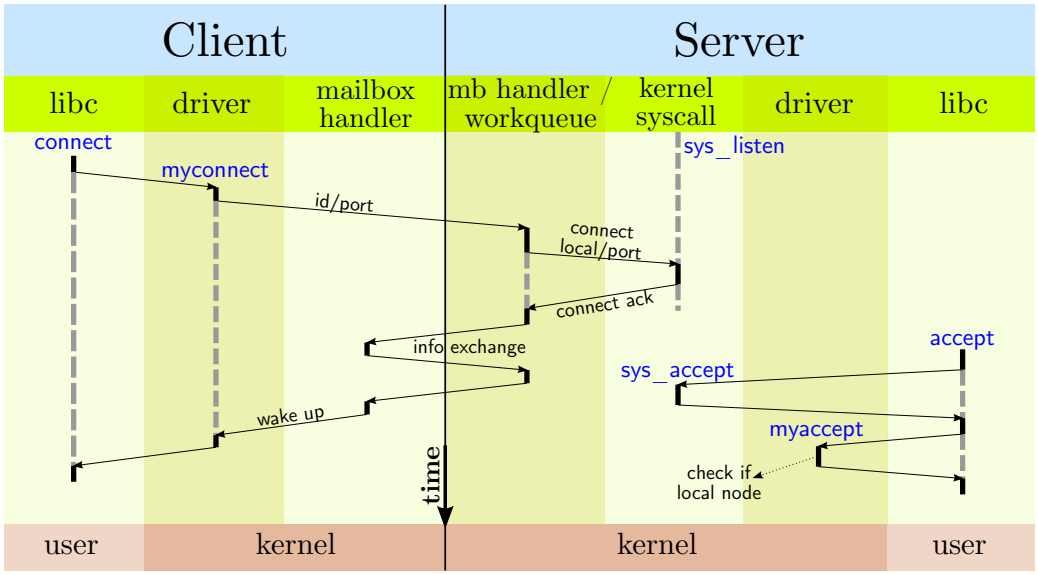

Figure 6.1: Connection establishment between local nodes

3. `Peer B` then, will follow the same procedure and choose its own ID, let's say number 5, which must be sent to `Peer A`. Now, for every message referring to this connection, each one will write the ID of the remote side as the message's connection ID. `Peer A` will send messages with ID 5, while `Peer B` will use number 3.

6.2.2 Connection data structures

Several data structures are used to keep the state of an open local connection. Some belong to the RDMA driver, while others are global variables in our modified `libc`. The most important of them are presented in the following paragraphs.

Connection tables

The RDMA driver has a table of active connections. Each element of the table is a pointer to a `connection struct`, which will be presented below. Its index number is the connection's ID. `Null` pointers denote available IDs. When a new ID must be obtained, the accompanying semaphore of the table is acquired and then the first empty element is used. Most calls to the RDMA driver refer to a connection, so this table is searched. This operation requires only a read to an element of the table and therefore does not need holding the semaphore. What happens when we close a connection and how elements get cleared will be discussed in Section 6.4. Except for the `connections table`, another data structure held in kernel space is the `newconnections structure`. We are going to describe this in Section 6.2.4.

In user space, connections are kept in a `libc connections table`. The dif-

64	56	48	40	32	24	16	8	0
0×C	localID		server port				nodeID	
0×A	remoteID			local buffer physical address				

0×A	remoteID	localID	local buffer physical address
0×A	remoteID		client port

Table 6.1: Client's & server's connection sequence messages

ference here is that the connection ID is not the index of the element but the element's value. In this table the indices denote a file descriptor number. Consequently, when an intercepted system call is issued, the file descriptor is checked in `libc connections table`. A value of zero indicates that the descriptor does not belong to a local connection (or is not a socket descriptor, at all). A positive value is a local connection. Because connection IDs begin from 0, the value stored in the table is actually $connID + 1$. Moreover, since it is very rare for a process to use more than a few dozen descriptors, the `libc connections table` has a length of 64 elements and then is continued as a linked list. Thus, traversing the whole list is rarely needed.

Connection struct

The `connection struct` is a structure used by the RDMA driver. All information regarding a local connection is gathered here: the ID of the remote peer, the remote connection ID, or the port and the remote port numbers. Moreover, additional information like the connection status (e.g. is it connected or still connecting) or the processes related to this connection (see Section 6.5.2) are stored here. Finally, details of the connection's local and remote buffers, that follow in the next paragraph, are part of this struct as well.

Connection buffers

Every connection has a pair of send & receive buffers on which all remote DMA operations occur. These buffers are allocated during the connection establishment phase and, as we will see in Section 6.3.1, are then mapped to user space as well. The driver keeps pointers to these buffers in the `connection struct`, whereas in libc there is another global table that resembles `libc connections table`, using file descriptor numbers as indices. The values of its elements are though, pointers to the mapped data. Both tables are updated at the same time.

6.2.3 Requesting a new connection

Initiation from client

When the client calls `connect`, our injected code checks if this destination belongs to the local network by reading the `peers` structure. If it does, the original system call is aborted and a negotiation with the remote (within the local network) side is initiated by exchanging a series of `mailbox` messages. In Table 6.1, the structure of these messages is shown. The first two are sent by the client and the other two by the server, in a *request / respond* fashion. All these actions take place in kernel space, by the two RDMA drivers.

Before contacting the server side, the client creates a new `connection struct` and finds a new `connection ID`. Then a `connection request (ConnRQ)` mailbox message is sent (first line in Table 6.1) and the client gets to sleep until a respond arrives. The `0×C` value on the left is the `message ID`, from which the server's `mailbox` interrupt handler will understand it is a connection request message. The connection request is the only type of message that its `connection ID`, the second field, does not refer to an ID belonging to the receiver of the message, but the sender. In other words, the client here says: "I want a new connection that I will identify with this ID". Of course, his `node ID` must also be included so the server knows who to respond to.

Dummy localhost connection

The last piece of information needed by the server is the `port number`, to associate the request with one his listening sockets. The difficulty at this point is that these sockets must be able to connect to both internal and external destinations. `listen` waits for incoming connections inside the kernel and we do not want to modify the kernel. We could intercept `listen` and have both the kernel and our driver waiting at this port number, the first waiting for network packets and the second for mailbox messages. Unfortunately, this way no arbitration of the connections could happen. Due to the multiple waiting sides, we would not be able to tell which connection is first in a case of close arrivals.

Therefore, when the `connRQ` message arrives a connection to the real socket is attempted. This connection is "dummy" because it is not actually going to be used, but serves only the purpose of confirming a connection to the real socket. The RDMA driver creates a new socket and issues a normal localhost `connect` to the same port. By localhost connection we mean a connection from within the node itself, via its *loopback network interface* which has the special IP address 127.0.0.1. Because `connect` is a blocking call, it cannot be executed from the interrupt handler[4] that received the message. So, this action is actually scheduled to the kernel's default workqueue[5].

[4]Sleeping inside an interrupt handler is not allowed

[5]Workqueues are special kernel mechanisms where tasks can be scheduled to run as kernel threads

Finishing the procedure

If the localhost connection succeeds, our connection procedure can be resumed. The server allocates a `connection struct` and gets its own connection ID for this connection. Furthermore, it allocates a pair of connection buffers. Afterwards, a response is sent to the client, that is seen in the third line of Table 6.1 (i.e. the first server message). From now on, every message sent by either side until the connection is established will be a `connection reply (ConnRPL)` type message, having a `0×A` as message ID. Additionally, all messages will contain the receiver's connection ID, in the connection ID field, so the receiver can recognize it. As a result, the server has to send his local connection ID , which the client is not aware of, on his first response.

At the client side the (intercepted) `connect` has not returned yet. The process is sleeping inside the call to the RDMA driver. When the server sends a respond, the client's interrupt handler knows which process has to wake up (which is in kernel space –the RDMA driver – and will remain there until the end of the connection procedure) because this information is already stored in the `connection struct` of the connection. On the other hand, at the server side everything is still done by the driver on its own and not on behalf of the server process. The `mailbox` interrupt handler receives the messages and continues the procedure (or schedules to the workqueue when it cannot sleep).

To finish the connection establishment procedure, the two sides exchange the physical addresses of their local connection buffers. More information about the structure and use of these buffers will be given in Section 6.3.1. Finally, the server acknowledges this last reception and then at the client side, the call to the RDMA driver returns the connection ID of the new connection, that is stored in `libc connections table`. `connect` can then return with a successful return value and the client is ready to use the connection from this point.

6.2.4 Accepting the connection

The server process is handed over a connection with a call to `accept`. Unfortunately, the same problem we faced with `listen`, applies to `accept` as well. We cannot intercept `accept` and ignore external accepted connections. To overcome this difficulty, the "dummy" localhost connection will also be used here. We will just let the server process accept it normally.

However, two issues arise with this approach. First, how can we identify our local connection? When `accept` succeeds, it returns a new socket descriptor for the connection. We need to know this number in order to intercept all upcoming data transactions. The solution here is to use *post-kernel interception* (Section 5.3.3). We let all `accept` calls run in the kernel and then before returning to the process, we match the new sockets with our localhost connected sockets. The second problem is synchronization; `accept` can be called at any time, before or after the client calls `connect`. When called before, as soon as the dummy localhost connection

succeeds, both `connect` of the workqueue and `accept` of the process will return. The latter should not be allowed to happen this moment because the process could then try to use the connection before our connection establishment has finished.

The `newconnections structure` is used to synchronize these events. Before any "dummy" localhost connection is issued, `newconnections` gets updated to show there is a pending connection. If it succeeds, it remains there, in a list of completed connections. On the other hand, when the real `accept` returns from kernel, our intercepted wrapper checks if the accepted address belongs to localhost. In this case, our driver is called and the `newconnections structure` is examined. If completed connections exist there, each one of them is also examined. Its pair is identified by the port number given to the "dummy" socket of the RDMA driver. However, `accept` may get there first, before `connect` has even returned. For this reason we keep the pending connections count. If no completed connection matches the accepted connection but there are pending connections, the `accept` side must wait for them to complete and then check them as well.

After a match of an accepted connection with a "dummy" socket, the rest of the connection establishment procedure must complete, if not yet, and then the intercepted `accept` can finally return to the process. Before this, the RDMA driver has returned the value of the new connection ID. Generally, because it is unknown whether `accept` will have to wait or not, *completions* are used. Completions are a kernel mechanism to wait on an event. If the event is already completed there will be no waiting. Ultimately, the dummy socket is closed and released, as it is not useful anymore. On the other hand the accepted descriptor is not closed since this could result in the kernel giving the same descriptor number to a new file.

One side effect of accepting local connections this way is that some irrelevant connections will be delayed with no reason. However, only accepted connections from localhost are examined. Furthermore, dummy connections are separated by port number in `newconnections` – here we mean the listening port. These two facts minimize the possibility of false waiting radically. For example, if the listening port is number 50000, only accepted connections from localhost will make the call to our driver and then in `newconnections` only connections for port 50000 will be looked. The matching criterion will be the other port number.

6.3 Data transfer

As soon as the connection establishment procedure has completed, the local sockets are ready to be used for data transactions. The two connected sides behave in the same manner from this point and the distinction between server and client is not applied anymore. Data transferring occurs when calling the pair of `send` / `recv` system calls or the more general `write` / `read`. Some variations of these calls exist, but there is not any substantial difference on the way that data are transferred.

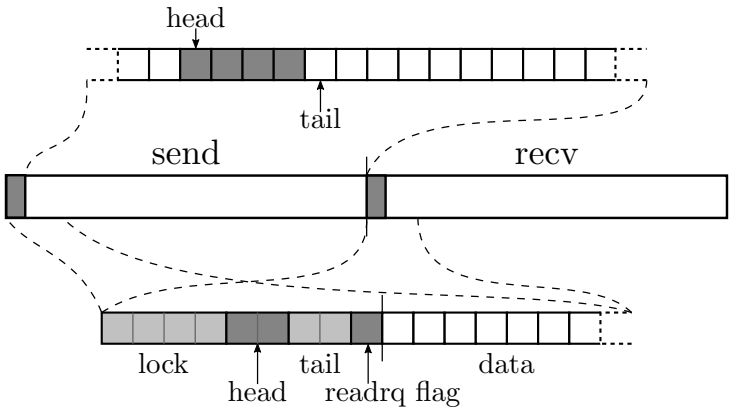

Figure 6.2: Send / Recv buffers organization

6.3.1 Send / Recv buffers

As it has already been mentioned, for all data transfers a pair of send and receive buffers exist in each end of a connection. We allocate *buffers per connection* to avoid synchronization problems among processes. As we will see, processes can modify these buffers from user space, so enabling accesses from multiple sources would require complex and time-consuming buffer protection.

The buffers come in two identical pieces, one for sending and one for receiving data, which are allocated during the connection establishment phase in contiguous memory space. Subsequently, their physical addresses are exchanged between the peers. Physical addresses are needed because the DMA engine operates with these. They are only stored in kernel space, within the `connection struct`.

Kernel memory is used for the allocation of the buffers and then they are memory-mapped in user space. This way the overhead of "pinning" the correspondent pages in memory is avoided. DMA operations use physical addresses and for this reason it must be assured that data source or destination pages do not get swapped out during a transfer.

The organization of the buffers pair of one side of a single connection is depicted in Figure 6.2. Apart from memory space holding data, there are also some special fields in each side. These are going to be discussed in the following paragraphs.

`head & tail` pointers

Both send and receive buffers of one connection are organized in a *ring buffer* (or circular buffer) scheme. This means that although they have physical ends, they do not have logical ends. Data reaching the end border can continue (wrap around) from the beginning, if there is adequate space.

By using ring buffers, data can be consumed and produced simultaneously, with each action occurring at each end. In our implementation it is possible that the user process stores more data to be sent, while at the same time the driver performs

a DMA operation sending older data. For this reason, ring buffers organization was followed.

Data boundaries inside a ring buffer are pointed by the `head & tail` pointers. The first indicates the position where data starts and the latter always points to the first free byte after the data. Any new insertion will happen there. This is shown in Figure 6.2, on the top. When both pointers have the same value, the buffer is assumed empty. According to the previous definition though, a full buffer would again have the pointers with the same value. To avoid confusion, one byte always stays free. Therefore, a full buffer (minus the one byte) has its head pointing at the next byte of its tail.

The location where the `head & tail` pointers are stored is on the buffer itself. Actually, a few bytes at the beginning of both `send & recv` buffers are holding special values and not data. This is illustrated in the lower part of Figure 6.2. Each pointer takes up two bytes of space, with the values stored there representing offsets within the buffer (from its start, not from the data's start). As a result, buffers up to 64 KB are supported with this configuration. The reason why these special "metadata" are stored together with the actual data, is that all these (data + metadata) must be shared between the RDMA driver and the user space process, as we will explain next. On the other hand, although each connected side knows the values of `head & tail` pointers of the remote `recv` buffer, these are only used by the driver and therefore are stored in the `connection struct`.

Read request (`ReadRQ`) flag

Except for the `head & tail` pointers, two other values are also kept in the beginning of each buffer. The `read request flag` and the `buffer lock` which we refer to in Section 6.5.

The `read request flag` is a single byte that can hold either 0 or 1. It is used only at the `send` buffer. When the flag is set (its value is 1), it means that the remote side has already sent a read request message to ask for new data and is actually waiting for them. As soon as new data arrive, they will be sent after checking this flag.

Mapping to user space

Data buffers are kernel buffers, also memory-mapped to user space. The benefit of this is that the user process can avoid calling the RDMA driver and undergo the user-to-kernel switch overhead. When data are already available in the receive buffer, they will be consumed immediately by a `read` call and when there is no need to send any data yet, a `write` will only store in the send buffer, hopefully coalescing data with other to come later. The use of buffers per connection allows this user space buffer access. Otherwise, one process would be able to watch data belonging to other processes.

As a matter of fact, this user mapping takes place at the end of the connection

Table 6.2: Read request message format

establishment phase. A call to mmap is issued right after returning from the RDMA driver in `accept` and `connect`, but before leaving the wrapper. `mmap` is a system call that creates new user mappings. For example, memory regions of a driver or even real files from disk can be mapped inside the memory space of a process. The real system call is issued – with no interception – and then the `mmap` implementation[6] inside the driver takes over. Most of the work though, like the page tables modification, is still done by the kernel.

The process calls `mmap` passing the driver's file descriptor as an argument. However, a connection ID must also be given: the ID of the newly created connection. The trick we do here is passing the ID via the `offset` argument of `mmap`. Then the driver performs the correct mapping and finally `mmap` returns a user pointer from where the buffers can be accessed. This pointer is stored in `buffers table`, which is another of our global libc tables, where buffers of all local connections of a process are kept.

Sharing resources between user and kernel space can lead to races with unpredictable results. For instance, the user process could read a value and then before it makes an action an interrupt could appear that stores something else, ruining the procedure. In our case, such risks do not exist because every important variable is always written by only one and because of the way the buffers are updated. For example, the tail of a send buffer is always written by the intercepted user space code of `write`, whereas its head is only updated by the kernel after a successfully completed transaction. Before the driver begins the RDMA operation, it reads both the head and the tail. The only thing that user space can do at this moment, is write new data after the old, that will not affect the current transaction. Data up to the old tail will be sent and moreover, the new data cannot overwrite the old, because the head pointer is controlled by the kernel side.

6.3.2 Data reception

With the current implementation, actual data transferring with RDMA operations are receiver-initiated. A normal `recv` call blocks until data are available, so the other side must be informed of this and unblock it as soon as possible. The `read request message` has been created for this purpose.

The format of `read request message` is shown in Table 6.2. The sender of the message also sends the current values of the `head & tail` pointers of the local receive buffer. With the current communication scheme, the remote side can

[6] drivers have to implement a specific function in order to support `mmap`; the same thing happens with `write`, `read` et al.

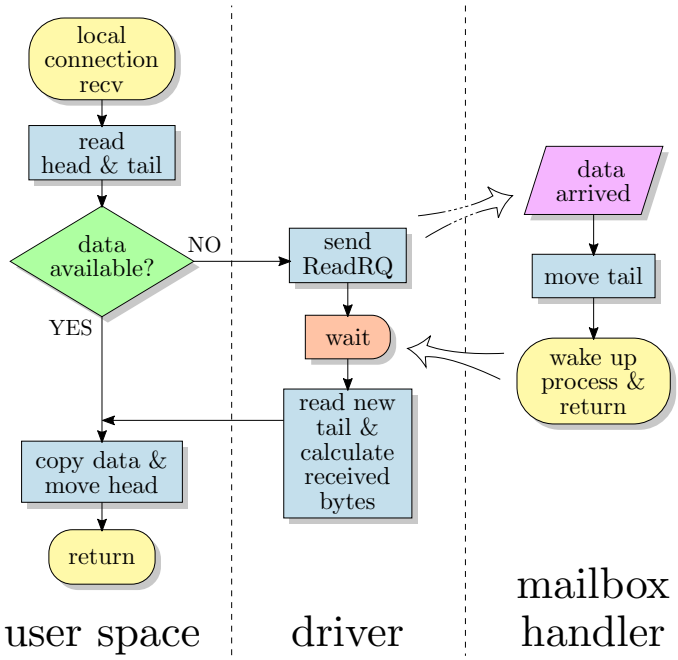

Figure 6.3: Data reception flow diagram

deduce the values of these pointers since it is the sole writer of the receive buffer and because the local side will not issue a read request before consuming all the previously received data. However, the pointers are included in this message as a safety measure and for possible future use.

There are two main parts playing a role in the receive procedure. The *synchronous* part, consisting of the intercepted system calls in `libc` and the RDMA driver in the kernel. Their actions are initiated by the user process, whereas the *asynchronous* part is the `mailbox` interrupt handler, that is executed depending on the data arrival moment. The `mailbox` interrupt handler is part of the RDMA driver, as well, but is not executed on the context of a user process. In Figure 6.3, a flow chart describing all actions performed at the receiver side is depicted.

Libc / RDMA driver

Generally, the intercepted `recv` performs the following steps when a local connection is encountered:

Check for data The first thing to do is check whether there are already data available in the receive buffer, by reading the `head & tail` pointers. This could happen if a previous `recv` had not consumed the whole available payload. If this is the case, the next step is bypassed.

Send a read request If `head` equals `tail`, then there are no data and a read request must be sent. The RDMA driver is called and the message is sent

by performing a simple store command to the remote `mailbox`. Afterwards, the process is put to sleep (in the kernel)[7], waiting to be woken up when data arrive. After the wake up, the tail pointer is read again to find out and return the number of received bytes to user space.

Consume data The available data or a part of them, if there are more than `recv` has requested, are then copied in the buffer given with the system call. Afterwards, they can be released from the receive buffer. This is succeeded by writing the suitable new head in the buffer.

As already stated, the receiver does not inform the other side on how many data he wants (the count argument of the system call), but how many data he can receive. The sender is free to fill the available space with as many data he has.

Mailbox interrupt handler

An interrupt is issued upon data arrival. After identifying the message type – it is a `remote interrupt message` that we will describe in Section 6.3.4 – and the referring local connection, the following are done:

Move tail Data are already copied in the correct position of the receive buffer, so now the buffer tail pointer has to be updated to show this. The number of received bytes are included in the interrupt message.

Wake up the process The driver knows the process associated with this particular connection, so the interrupt handler reactivates it[8].

The big arrows in Figure 6.3 represent that the interrupt is eventually caused by the read request message sent earlier. Consequently, in this case there is always a process waiting for this interrupt.

6.3.3 Sending data

The flow chart presenting the steps performed at the sender's side to send data to a local connection is split in Figure 6.4, for the `libc` / driver part and in Figure 6.5 for the `mailbox`.

Libc / RDMA driver

In order to send data to the remote side, a `send` call has to be made. The time of actual sending through the network though, is not always known.

[7] special care has to be taken to avoid the *lost wake-up problem* [7]. This happens when a wake-up event arrives after it was decided to put the process to sleep, but before this procedure is completed. As a result, the event is lost.

[8] this includes changing the process state to `TASK_RUNNABLE` and placing it in the task scheduler's list

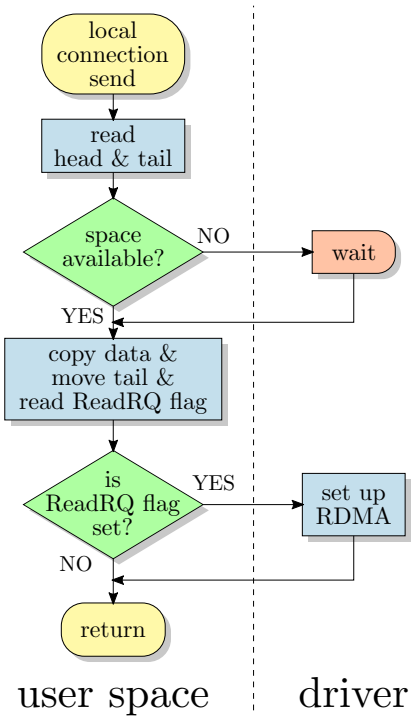

Figure 6.4: Flow chart of libc / driver when sending data

Compute available space Initially, the available space in the send buffer must be checked. This is deduced by the `head & tail` pointers.

Wait if buffer is full If the send buffer is full of unsent data, the system call has to block until some space is freed. As usual, the process sleeps in the driver.

Copy data Subsequently, the correct amount of data is copied to the send buffer and then the tail pointer is increased respectively.

Check ReadRQ flag & perform RDMA The `read request flag` has to be checked at this point, because if a request had arrived earlier the transaction would not happen. If the flag is set, the RDMA driver is called to perform the RDMA operation.

Mailbox interrupt handler

There are two types of interrupts affecting the `send` procedure. It will be described below how each type is handled:

- **Arrival of a read request** (left side of Figure 6.5)

  Check for data The `head & tail` pointers are read and then one of the next two steps is performed.

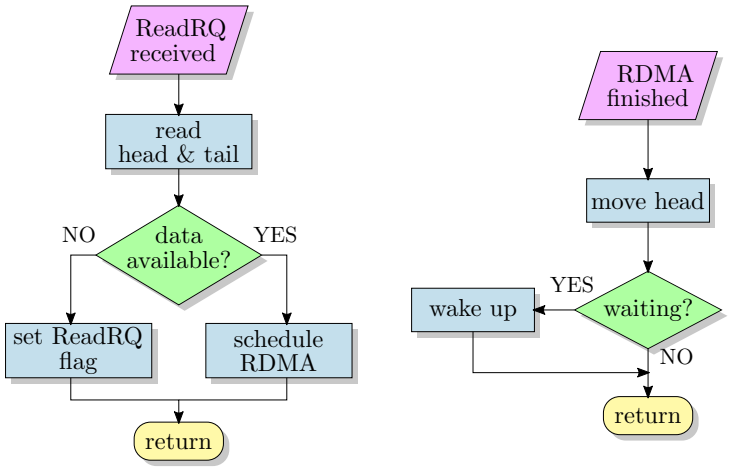

Figure 6.5: Mailbox interrupts affecting the sending procedure

Set the ReadRQ flag If the send buffer is empty, the first following `send` system call has to know that the remote side is waiting for data and send them immediately.

Schedule RDMA If there are data to be sent, an RDMA operation must occur. Unfortunately, the interrupt handler cannot perform this action (it may sleep), so it is scheduled in the kernel's default workqueue to happen as soon as possible.

- **Completion of an RDMA operation** (right side of Figure 6.5)

  Move head When an RDMA operation finishes successfully, the send buffer has to be updated to free the reserved space.

  Wake up process If the process is blocked on a `send` call, it can now be unblocked since there is free space in the send buffer again.

The arrival of a `read request` can occur between the two events of moving the tail pointer and checking the read request flag in the wrapper. This rare situation, however, cannot lead to an erroneous outcome. The handler will see that data exist and will schedule a transaction, without setting the flag. Consequently, the code in the wrapper will not begin another RDMA operation. Additionally, no extra read request can be delivered, because the remote side is blocked, waiting for data.

6.3.4 RDMA operation

Making an RDMA operation requires preparing the descriptors that are going to be used and writing the last of them to the `TAILDESC_PNTR` field of `CDMA`. To perform these two actions the descriptors semaphore must be held and as a consequence,

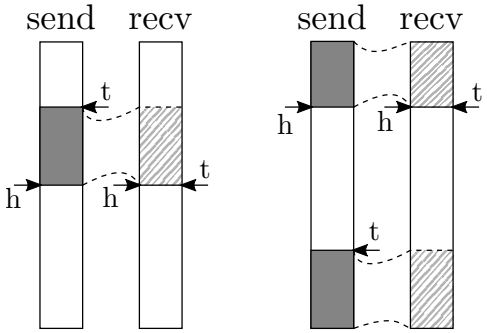

Figure 6.6: Data transferring with 1 and 2 transactions

this procedure may sleep in the case of a contended semaphore. This is why the `mailbox` handler schedules an RDMA rather than carrying it out at once.

Data transferring

In the general case, copying data between two ring buffers normally requires from 1 up to 3 different transactions, depending on the data position at each side (if they wrap around). In our case, because the receive buffer of one side always follows the send buffer of the other, only 1 or 2 transactions may have to be performed. The latter case occurs when data expand beyond the physical end of the buffer to its start. The first transaction will copy the data from the `head` pointer, up to buffer's end and the second from its start, up to the current `tail`. Figure 6.6 shows these two cases.

Local & remote interrupt

After a successful RDMA operation, both connection sides have to be notified. The remote side, in order to be unblocked from the `read` call and to wake up the local process possibly waiting for space in the send buffer.

The only way to perform a *remote interrupt* is by using the `mailbox` mechanism along with the address translation feature. A simple (64-bit) store to the address of the remote `mailbox` will trigger the interrupt, which will also be secure against other simultaneous interrupts possibly coming from other nodes, due to `mailbox`'es fifo list.

As far as the *local interrupt* is concerned, the interrupt caused by `CDMA` could be used. However, this is not easy because of the way this interrupt is triggered. When operated in `scatter gather` mode, `CDMA` produces an interrupt after a programmable but also fixed number of completed descriptors. Unfortunately, as we have mentioned above, transactions needed can either 1 or 2. Except for this, we would also have to identify the connection that the interrupt belongs to. For these reasons, we use the mailbox again to produce the local interrupt as well.

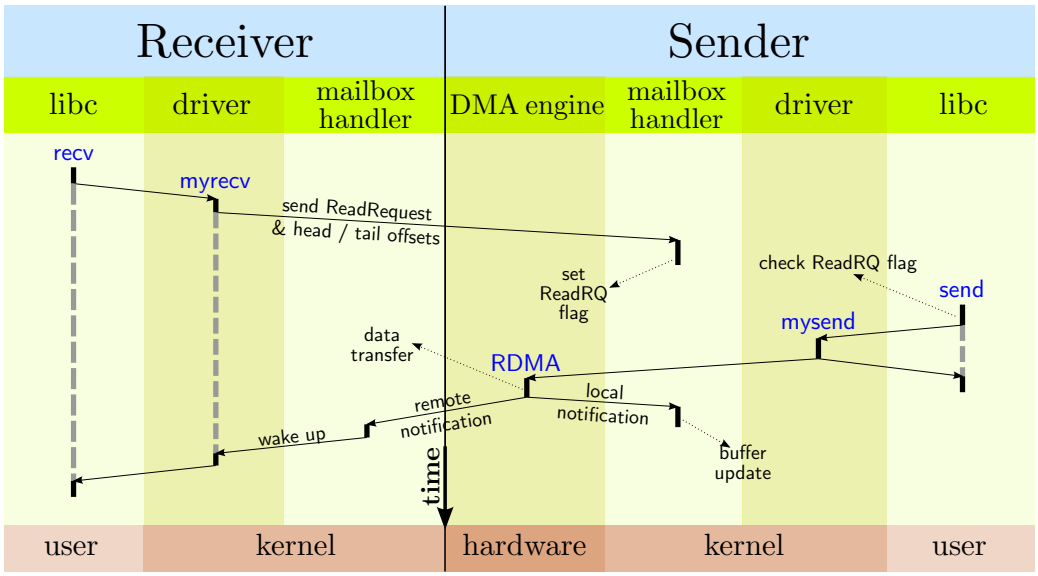

Figure 6.7: Sending and receiving data

Descriptors

Eventually, a complete RDMA operation consists of 3 or 4 descriptors: The 1 or 2 data copying descriptors and the other 2 producing the local & remote interrupts.

Data descriptors are straightforward to be prepared. On the other hand, for the interrupt descriptors, the DMA engine needs to read the content of the mailbox messages from somewhere. For this, we take advantage of the free 32 bytes after each descriptor (Section 6.1.3), where the 8-byte messages are written. Their format is illustrated in Table 6.3.

■

An example of a complete `send` / `recv` procedure is given in the timing diagram of Figure 6.7. The DMA engine is also added in this diagram, with the thick line indicating the data transfer in this case. In this example, `recv` has been called before `send`. Furthermore, only one `send` is issued, so there could not be a case of blocking in a `send` call.

64	56	48	40	32	24	16	8	0
0×F	localID						bytes	
0×B	remoteID						bytes	

Table 6.3: Local & remote interrupt mailbox messages

6.4 Closing a connection

A socket is terminated either by explicitly calling the `close` system call in the user code or when the process ends.

6.4.1 Freeing resources

In our injected code in `libc`, there are no dynamically allocated resources to be released. A `close` call, updates our global tables to forget this local connection and closes the real accepted socket descriptor, which was left open, so that the kernel can reuse its number. However before this, a call to the RDMA driver is also needed to release the connection resources allocated in the driver. These include the local buffers and structs like `connection struct`. Finally the driver's global `connection table` must also be modified.

If `close` is not called at all, the driver cleanup still takes place with the help of the kernel's cleanup procedure of the user process. The kernel will call two registered cleanup functions of our driver. The first, when the mapping created by `mmap` is undone and the second when eventually releasing the driver from the process. As a result, the connections belonging to that process are again normally released preventing memory leaks in the driver's memory.

6.4.2 Disconnecting from the remote side

What happens though, with the remote connected side? If a connection is simply closed unilaterally, several problems can emerge, either affecting the remote or the local side.

Possible situations

First of all, the remote side could have been waiting for data and consequently, would be stuck there forever. This is easily handled: During the connection cleanup, it is checked whether there is any read request already. If this is true, a `remote interrupt message` is sent with a value of 0 in the bytes field, causing `read` to also return 0. This is exactly what `TCP / IP` does in a similar case. If the request arrives after the closing procedure has completed, this message is sent by the interrupt handler. On the other hand, if the closing side has any data in the send buffer, these are sent normally and the cleanup continues thereafter.

Another possible difficult situation occurs when the remote side is preparing an RDMA operation and the connection is closed locally. This could result in overwriting memory not belonging to the driver anymore, risking the whole system's integrity. An RDMA happens only if the local side is waiting for it and as a result, the connection could be closed safely if this is not the case. Unfortunately, if it is already waiting though, there is nothing than can be done. It cannot be guessed whether and when the remote side will send data. On this occasion, some kind of negotiation must be done between the two sides.

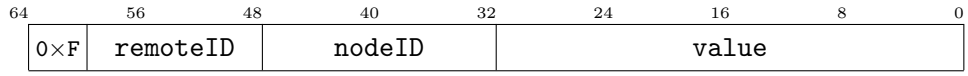

Table 6.4: Format of `NACK` mailbox message

Finally, another case showing that this negotiation is inevitable is the following: the local side closes a connection without any of two above problems happening. The connection ID is then released and after a while it is used again for a connection with another peer. However, the original connected peer still thinks he is connected and can send messages at any time; the local side will not be able to recognize from whom the messages came from.

The `NACK` mailbox message

The `NACK` mailbox message is used to abort an internal network connection by informing the remote side before any actual cleanup. This is not enough, however. The aborting side must be sure that the message was delivered and handled. As a result, a *disconnect procedure* actually takes place.

The two sides perform a *handshake* using `NACK` messages. The remote side receives the first `NACK` and cancels any upcoming, or waits any ongoing RDMA operations. Afterwards, it responds with another `NACK` message and when it is delivered, the other side can finally release the connection's resources. After a timeout, the original aborting side would resend the message if there was no respond from the remote side.

The format of the `NACK` message is depicted in Table 6.4. The sender can also include an integer value in the message representing the reason for aborting. Furthermore, his node ID must also be sent, handling situations like the following one: Assume that Peer A first send a `NACK`. Peer B receives it but sends his response delayed, so that a resend from Peer A happens. However, as soon as Peer B sends his response, he considers the connection as over and could have even started a new one with the same ID. Therefore, after receiving the new `NACK` the node ID in the message will clarify who sent it. Beginning a new connection with Peer A using the same connection ID is not problem, because Peer A keeps a list of uncompleted `NACK` procedures.

6.4.3 Abort while connecting

Aborting the connection establishment procedure is again carried out with `NACK` messages. This could occur in a case of a system error (e.g. out of memory error), a response timeout, or the user interrupting the procedure. On this occasion, the `value` field of the message is also used to give the reason, using standard defined values of the Socket API like `ECONNREFUSED` or `ECONNABORTED`. These values are then returned from the system call of the remote side.

Additionally, a special case exists that requires different handling. If a node sends a `CONN_RQ` message to begin a connection and the host does not respond for some reason, or if the client decides to abort before the first reply, then the client does not know a remote connection ID yet to send a `NACK`. In this situation, `connect` returns to the user with an error code, but the driver holds the ID for a longer period, until it can presume that the connection has failed.

6.5 Multithreaded & forked application support

Multithreaded and forked applications result in sockets being shared among different processes or threads. This sharing could easily cause many problems to our system. For example, a process issues a `read` and afterwards, a child process issues another `read` on the same socket. How will two read requests be handled? Who will eventually get the data in this case? The RDMA driver could have been designed to serialize such accesses, but unfortunately, due to the mixed user & kernel space architecture of our system, this would not be enough.

It has to be noticed, however, that normal socket programming does not involve concurrent use of sockets. For example, a server usually accepts a new connection and then creates a new thread to deal with it; no other thread will use this socket. Or in forked applications, usually either the parent or the child keeps a socket, while the other one closes it right after the fork takes place. However, since it can happen, it must be handled. The same thing is done by the kernel network stack. It ensures that all data are transferred, even mixed with each other sometimes.

Therefore, we have implemented *custom locks* to protect socket operations on our internal network. In general, the RDMA driver is designed to prevent events that could compromise the integrity of the whole system (the OS and other running applications), whereas data integrity of our local sockets is handed over to this locking procedure in `libc`.

6.5.1 Custom locks implementation

We could not use the mutexes offered by `libpthread` because our injected code – where the locks are needed – is part of `libc`. Some elements of `libpthread` are also integrated in `libc`, but features like shared mutexes among processes – necessary for forked applications – are only found in the first. So, `libc` would have to be linked to `libpthread` and this would require significant changes in the build system of these libraries. Apart from this, we need the locks to specifically lock connections, so our RDMA driver can be directly used, rather than employing the kernel `futex` subsystem that is more generic[9].

Our locks are implemented in a similar way to `pthread mutexes`. A memory location is used to perform *atomic operations*[10] to. The first to modify this location

[9] pthread mutexes use the `futex` system call to wait on a particular memory location; all these locations are kept in a hash table in the kernel so processes can sleep or woken up

[10] we use the atomic `compare and swap` provided by `gcc`

gets the lock. Others have to call the driver to wait. Before this though, they modify – again atomically – the location so that it now indicates that not only it is locked, but contended, as well. Afterwards they are put to sleep in the driver[11]. When the lock owner releases the lock, he calls the driver to wake up one, if it is contended.

What needs to be locked in our case, are the different connections. Each connection has buffers and these buffers are shared among all users of it, the RDMA driver included. Consequently, it is convenient to place the locks in these buffers, as we have already seen in Figure 6.2. Furthermore, because every connection has two independent paths, one for sending and one for receiving (and hence two buffers per side), two locks are used per connection.

6.5.2 Tasks list

The RDMA driver keeps a list of all processes that use a particular connection. This is a linked list inside every `connection struct`. Its elements are also structs, including a pointer to the process along with information referring to its current state. This pointer has the type of a kernel struct called `task_struct`, as processes are called tasks in kernel terminology. An interesting thing is that threads are also represented with the same struct and are in fact, actual tasks. They differ from normal processes in that they share many parts with other tasks. As a result, the `tasks list` of a connection can include threads as well.

The state we keep for every task is its state in relation to the connection. That is for example, if it is waiting on a `recv` call, or on the send buffer lock, etc. This information is used by the driver to wake up the correct tasks. Except for this, the other reason that the tasks list is required for, is security. Most of the calls to the driver also contain a connection ID. One of the first things done by the driver when serving these calls, is to check out if the calling process has indeed access right for the connection. Otherwise, it would be easy for any user process to steal data from others.

However, because the size of the `tasks list` could become very large, while usually one only task is using the connection, there is actually a second list. The `tasks list` contains all the legitimate potential users of the connection, while at the `users list` only the actual users of it are inserted.

6.5.3 Using the locks

The usage of locks is straightforward. For every system call which involves sending data (`send, write, sendmsg` et al.), the intercepted call acquires the send lock of the connection before doing anything and releases it at the end. In the same manner, all receiving system calls lock the receive path until getting and consuming the data.

[11] again the lost wakeup problem (Section 6.3.2) must be dealt

For the case of `connect` and `accept`, locking primitives are not needed. Concurrent `connect`s will safely – due to driver's locking – create multiple new connections, whereas with `accept`, again different connections will be accepted, with the kernel this time handling concurrency.

Finally, locks are being used in the intercepted `close` system call, but this is done to protect the procedure of `dup`, for which we are referring in Section 6.6.1.

6.5.4 Cloning technique

In Linux, creating threads and forking processes are actually implemented with the same system call, `clone`[12]. The difference between these two is the amount of resource sharing between parents and children. For example, threads share memory space, while in `fork` a *copy-on-write* technique is employed.

The existence of our custom locks would suffice for running multithreaded and forked applications, without any interception at the cloning procedure. However, we have also intervened there, for the two reasons following.

The `clone table`

We want to avoid the overhead of locks, when they are not needed. Singlethreaded applications would always run with uncontended locks, not entering the kernel, but still the usage of atomic primitives wastes many CPU cycles.

For this purpose, another global table has been added in our modified `libc` and `libpthread` code. The `clone table` uses file descriptor numbers as indices, like the table of local connections, and gets updated after cloning procedures. The values of this table show if a particular local socket needs locking or not.

To give an example, let us assume that a process has a local connection, to which the file descriptor 3 is assigned. If the process forks itself, both the parent and the child will now have the value of 1 at the 4^{th} element of their `clone table`. This value instructs any data transferring operation to use the locks, because the connection is shared. On the other hand, if the child later creates another local connection, its value on the table will remain 0, as the parent has nothing to do with it. Therefore, a `send` call will see this value and will not acquire or release the lock to send the data.

Updating `tasks list`

Earlier, we mentioned that only tasks included in the driver's `tasks list` of a connection can use the connection. But how does this list get its elements? The first user is simply added during the connection establishment procedure. When cloning occurs, the new child has to be announced to the driver.

[12]For compatibility, the old `fork` and the `vfork` variation still exist as separate system calls, but in the kernel they still use the subsystem of `clone`

One possible solution, could be the child to call the RDMA driver, which could confirm its parent identity, using the kernel struct describing the child. However, it is possible that the parent has already terminated before this. In this case, the pointer to the parent at the kernel's struct would be `NULL`. The exact same thing could happen, if the parent was the one to enter the driver to announce its child. Consequently, to solve this problem, both of them call the RDMA driver, with the first waiting the second. Thereafter, the `tasks lists` of all local connections the parent has, are updated.

An interesting detail is that we do not use a lock to protect a read of the list from a possible update of it – concurrent updates are not allowed, however. Traversing the `tasks list` occurs very often and this would cause overhead. In fact, we use the double-ended linked lists provided by the kernel and we only add elements in the end. The way that the kernel function of adding an element is written, could only ruin the reverse traversal of a list. Since we use only use the normal direction, each read of the list is protected.

6.6 Other supported features

6.6.1 The `dup` family

The `dup` family of system calls (`dup, dup2, dup3`) is used to create *aliases* of file descriptors, including socket file descriptors. For example, a `dup(3)` call that returns 4, has created a new file descriptor (number 4), that is exactly the same with the old descriptor (number 3).

Intercepting a `dup` call referring to one of our local sockets is simple. We let the real system call run in the kernel and intercept its result. This is another case of *post-kernel interception* (Section 5.3.3). The file descriptors that are passed to the kernel as the argument of `dup`, are the original socket created by the `socket` call at the client, and the socket created by the real `accept` at the server. The kernel thinks that the first is still unconnected and that the other had a localhost connection and is now waiting to be closed. Both of them though, are not closed, so their descriptor numbers are still valid. In our previous example, our intercepted call will just copy the value of the 4^{th} element of `libc connections table` and `buffers table` to the 5^{th} element. Now both these file descriptor numbers will refer to the same local connection.

The problem here is with `close`. Having duplicated descriptors means that only the last `close` call, actually closes them. All previous just remove the aliased descriptor numbers. As a result, another table has to be maintained, to keep the count of descriptor numbers assigned to local sockets. The `socket_count table` has elements with indices referring to `connection IDs`. Their values are the count of each connection and are updated while holding the send lock – the recv lock could also be used – to prevent errors.

6.6.2 Socket options

There are lots of socket options, controlling various parameters of socket connections. Currently, only the most common of them is supported, the `SOCK_NONBLOCK`. This is actually an attribute of file descriptors and not only sockets. It can be set either in the `socket` call, or later using `fcntl`.

The non-blocking attribute makes all calls to a socket return immediately. For example, if no client has connected yet to a server, `accept` returns an error and so does a `recv` when no data are available.

A noteworthy change that was implemented when dealing with non-blocking local sockets is the following: At the sender's side of a data transaction, a `send` would never block; if the send buffer is full, an error is returned. Therefore, the local interrupt is useless in this case. What we do is replace the RDMA descriptor causing the local interrupt, with a new that only moves the head of the local send buffer. This way, we avoid the interrupt handling overhead.

Chapter 7

Evaluation

In this chapter we will present some evaluation results of our system and compare it to the typical TCP/IP network stack. This comparison, however, is not very fair, as the two sides use physical networks with different capabilities. Throughput results are affected more by the network capacity, whereas for latency measurements we could claim that the protocol induced overhead is uncovered to a greater extent.

Our internal network employs the custom interconnect of the prototype, that was described in Chapter 4. On the other hand, for the TCP evaluation, the two boards were connected through their onboard 1 Gbit network interfaces, on a back-to-back configuration using a crossover ethernet cable.

7.1 Evaluation benchmarks

Several microbenchmarks were created to confirm correct operation and measure our system. Additionally, real world applications like the network instrumentation utility `iperf` [8] were used. Two main tests that allow us to evaluate our internal network's capabilities have been carried out: the latency and the throughput test.

Latency test

The first network parameter that we measure is latency. This includes the latency imposed by the physical link and the latency added by our protocol and the operating system. Our test is a microbenchmark that performs a *ping-pong* style communication between the two peers. One of them initially does a `send` operation, while the other waits for the data with a `recv`. As soon as this transaction is completed, another one is carried out in the opposite direction this time.

These transactions do not overlap, because each time one is sending and the other is waiting. As a consequence, if we measure the total time that one side needs to perform these two actions we have the aggregate latency of the two transfers. Supposing they are symmetrical, the half of this time is the latency we want: The latency from the time the sender starts sending data until the receiver consumes

		Latency ($\mu Secs$)	
		RDMA	TCP / IP
Transfer size (Bytes)	16	13.99	62.35
	32	13.97	62.49
	64	14.04	62.89
	128	14.58	64.38
	256	14.80	67.18
	512	15.29	72.54
	1024	17.33	84.78
	2048	21.17	85.11
	4096	28.06	85.44

Table 7.1: Latency evaluation

them. To get more accurate results, this measurement is performed hundreds of thousands of times. The connection establishment overhead is not part of the measurement.

Throughput test

To test the throughput capability of our system, the `iperf` utility was used. This test consists of one side constantly sending data, while the remote side receives them. The total amount of bytes divided by the time elapsed for the complete transaction, gives the throughput result. Measurements with different transfer sizes and different number of concurrent connections were done. Using the `--parallel` (or `-P`) option of `iperf`, several threads are spawned, with each having its own connection and performing the test. Here, of course there is no case of contended locks, as these threads act exclusively on their own connections.

Due to the mixed user & kernel space design of our system, data transferred are heavily coalesced for small message sizes, thus utilizing the network more efficiently. The intercepted system calls do not constantly call the RDMA driver, but take advantage of the connection's buffers to store or read data from.

7.2 Evaluation results

In Table 7.1, latency measurements for different sizes of transfers (in bytes) are shown. The results are visualized in Figure 7.1. Comparing to the TCP / IP latency, our Sockets over RDMA system is 3 to 5 times better. We can see that for small sizes, data copying and transferring is not so important and latency is mostly determined by factors like the interrupt handling or the user-to-kernel and kernel-to-user switch overheads.

Throughput measurements are given in Table 7.2 and in Figure 7.2. Here, the

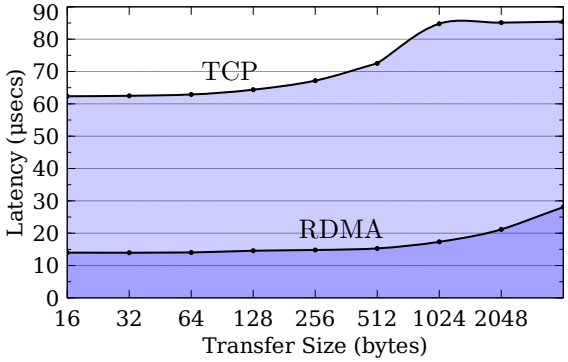

Figure 7.1: Latency evaluation

		Throughput per parallel connections (Mbps)							
		Sockets over RDMA				TCP / IP			
Connections		1	2	4	8	1	2	4	8
Transfer size (Bytes)	16	556	521	425	249	17	42	40	42
	32	1018	922	731	426	37	86	89	86
	64	1329	1069	851	519	94	152	155	155
	128	1425	1441	1036	590	121	239	239	236
	256	1438	1522	1144	621	148	346	316	306
	512	1425	1587	1188	648	189	383	402	399
	1024	1440	1642	1282	662	232	483	467	425
	2048	1435	1610	1296	698	269	544	451	427
	4096	1437	1634	1321	717	302	558	439	428

Table 7.2: Throughput evaluation

comparison between our internal network and TCP / IP is not valid, because of the different link capabilities. However, we can contrast the behaviour of the two systems in terms of scaling to multiple connections, while using different transfer sizes. As it can be seen our network rapidly exploits the link's limits from small sizes, by taking advantage of the data coalescing feature. On the other hand, it does not scale well as the number of connections begins to grow. An explanation for this is probably the contention that RDMA descriptors begin to have. As we have referred to in Section 6.1.3, all connections use a common set of descriptors that must be locked to avoid corruption when preparing a transfer.

7.3 Analysis of overheads

We have used the special registers in the *Performance Monitoring Unit* of the `A9` processor, that count CPU cycles [9, 10], to profile our system's behaviour. This

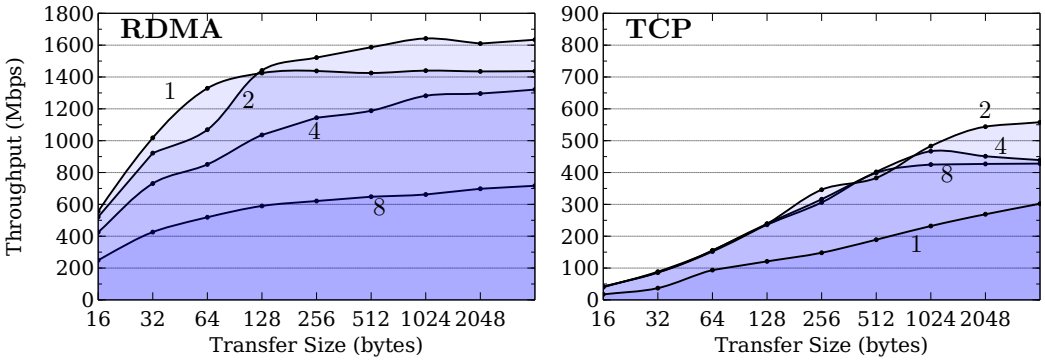

Figure 7.2: Throughput evaluation for 1,2,4,8 parallel connections

way, we can obtain accurate results and analyze the cost of every software and hardware component to the total latency.

Connection establishment

The time needed to establish a new connection is 100 μsec, compared to the 180 μsec of TCP. While it was not our primal concern, creating new connections rapidly can benefit significantly modern servers, which often service numerous short-lived clients.

Of course, the dummy TCP localhost connection is also included in these 100μsec. Still, our connection establishment sequence is faster though. Our measurement takes place at the client side and is essentially the time that the `connect` system call blocks.

Latency breakdown

Again, by using the special cycles registers of the processor, several measurements have been made in order to analyze and rationalize the latency we found before (Table 7.1). Here, we will only present the numbers for 16-byte transfers. However, many of the following latency factors do not depend on transfer size and are always the same. The latency breakdown for the total of 14 μsec is illustrated in Figure 7.3 and is described in the following:

- On the sender's side, the time spent in user space, in our injected libc code, before entering the kernel is around 0.8 μsec. During this interval, data are copied to our buffers and thus, this value depends on the number of bytes.
- A context switch from user to kernel space costs approximately 1.8 μsec
- After entering the kernel, the RDMA driver needs about 0.9 μsec to perform some checks before initiating the transfer.

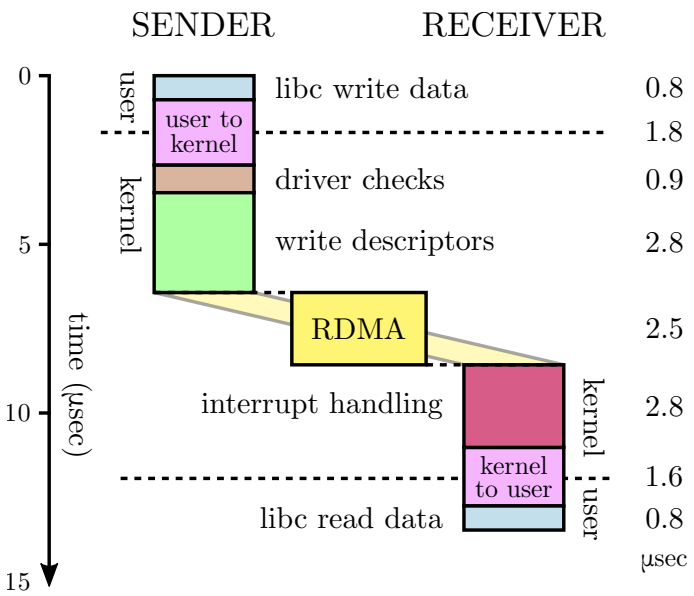

Figure 7.3: Latency breakdown of a 16-Byte transfer

- Afterwards, to prepare the RDMA descriptors, 2.8 μsec are spent. This number reflects the preparation of 3 descriptors, which is the common case.
- The data transfer costs around 2.5 μsec. Again, for larger transfer sizes, more time is needed.
- The time required to serve interrupts on the receive side is about 2.8 μsec.
- The kernel-to-user context switch takes 1.6 μsec.
- After returning from our driver, 0.8 μsec are needed to copy the 16 bytes to the buffer supplied by the receiver's user code.

We begin this breakdown from the sender's side and end it on the receiver, as is easily seen in the figure. One important detail of the 16-byte transfer is that the interrupt handling interval we have mentioned, concerns 2 interrupts on the receive side. After the data transfer has completed, the local interrupt is sent to the sender and the remote interrupt to the receiver. However, due to the delay of the interconnect, the local one is delivered first. As a result, the sender returns from the `write` call and continues to the subsequent `read`. This causes a read request to be sent to the receiver. We have verified (by using counters in the driver's code) that 99% of the times the read request is served together with the remote interrupt[1].

[1] After completing a particular interrupt, the interrupt handler always checks if a new interrupt has arrived, to avoid the overhead of exiting and reentering the interrupt context.

Chapter 8

Conclusions and Future Work

In this work, we have presented a mixed user & kernel space software architecture, implemented to allow unmodified applications to communicate over a remote DMA capable interconnection network. For this purpose, user space interception of system calls related to the `Socket API` was employed within a modified `Standard C Library` environment.

Basic functionalities like creating TCP connections and performing data transactions through them have been implemented, along with support for more advanced features like multithreaded and forked applications. Development and testing has been carried out on the first version of `Euroserver Discrete Prototype`, to explore potential benefits of accelerating internal communication in a microserver environment.

This effort has a lot room for improvement either it is a better support of the `Socket API` or optimization of our implementation to achieve better performance. For the first case, more complete coverage of system calls and socket options can be explored. System calls like `select` / `poll` or `sendfile` are not handled yet, whereas support for more socket options, will allow applications to fine tune the network parameters and capabilities. Furthermore, the connection establishment procedure can be altered, as in its current form, it relies on a real TCP localhost connection. This imposes the restriction of having always to bind listening sockets to `0.0.0.0`.

On the other hand, several techniques could be tried to achieve better performance and usability of our system. Making transactions with *Zero Copy* capability, would vastly improve the performance of large message communication, both in terms of latency and throughput. Extending the current communication scheme could also turn out beneficial. For instance, a `write request` message could be employed to allow the sender to initiate data transactions. Moreover, better buffer management could also make a difference. A more dynamic buffer management scheme with variable buffer sizes or shared buffers would save valuable memory space.

Bibliography

[1] Intel Corporation, "Flexible, low power microservers for lightweight scale-out workloads," Intel Newsroom, 2013. [Online]. Available: http://www.intel.com/newsroom/kits/atom/c2000/pdfs/Intel_Microserver_Whitepaper.pdf

[2] N. Heath, "Microservers: What you need to know," ZDNet, April 2014. [Online]. Available: http://www.zdnet.com/article/microservers-what-you-need-to-know/

[3] M. Kerrisk, "Linux man pages. Section 2: System Calls; Section 7: socket, ip, tcp, udp, raw," Linux man pages online. [Online]. Available: http://man7.org/linux/man-pages/

[4] Xilinx, Inc, "LogiCORE IP AXI Central Direct Memory Access v3.03a - Product Guide," Xilinx online documentation, October 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v3_03_a/pg034_axi_cdma.pdf

[5] C. O'Donell, "System call wrappers," Glibc Wiki, April 2013. [Online]. Available: https://sourceware.org/glibc/wiki/SyscallWrappers

[6] ARM Holdings plc, "Procedure Call Standard for the ARM® Architecture," ARM Infocenter, November 2012. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf

[7] K. Sovani, "Kernel Korner - Sleeping in the Kernel," Linux Journal, July 2005. [Online]. Available: http://www.linuxjournal.com/article/8144

[8] "Iperf network testing tool." [Online]. Available: https://iperf.fr/

[9] ARM Holdings plc, "Cortex-A9 Technical Reference Manual: Performance Monitoring Unit." [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0433b/CIHJGICA.html

[10] "User-mode performance counters for ARM/Linux." [Online]. Available: http://neocontra.blogspot.gr/2013/05/user-mode-performance-counters-for.html

[11] 6WIND & Intel Corporation, ""Intel DPDK: Data Plane Development Kit"." [Online]. Available: http://dpdk.org/

[12] ntop, "PF_RING: High-speed packet capture, filtering and analysis." [Online]. Available: http://www.ntop.org/products/pf_ring/

[13] "OpenFabrics Alliance (OFA)." [Online]. Available: http://www.openfabrics.org

[14] Infiniband Trade Association, "InfiniBand™Architecture Volume 1 and Volume 2." [Online]. Available: http://www.infinibandta.org/content/pages.php?pg=technology_public_specification

[15] R. Recio, B. Metzler, IBM Corporation, P. Culley, J. Hilland, Hewlett-Packard Company and D. Garcia, "A Remote Direct Memory Access Protocol Specification," RFC 5040, October 2007. [Online]. Available: http://tools.ietf.org/html/rfc5040

[16] Infiniband Trade Association, "Supplement to InfiniBand™Architecture Specification Volume 1 Release 1.2.1: Annex A16: RDMA over Converged Ethernet (RoCE)." [Online]. Available: http://www.infinibandta.org/content/pages.php?pg=technology_public_specification

[17] "IEEE 802.1 Data Center Bridging Task Group." [Online]. Available: http://www.ieee802.org/1/pages/dcbridges.html

[18] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1991596.1991611

[19] N. Provos, "Improving Host Security with System Call Policies," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251371

[20] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker," in *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, ser. SSYM'96. Berkeley, CA, USA: USENIX Association, 1996, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267569.1267570

[21] K. Scott and J. Davidson, "Safe virtual execution using software dynamic translation," in *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, 2002, pp. 209–218.

[22] U. Erlingsson and F. B. Schneider, "SASI Enforcement of Security Policies: A Retrospective," in *Proceedings of the 1999 Workshop on New Security*

Paradigms, ser. NSPW '99. New York, NY, USA: ACM, 2000, pp. 87–95. [Online]. Available: http://doi.acm.org/10.1145/335169.335201

[23] S. H. Rodrigues, T. E. Anderson, and D. E. Culler, "High-performance Local Area Communication with Fast Sockets," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '97. Berkeley, CA, USA: USENIX Association, 1997, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268680.1268700

[24] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 256–266. [Online]. Available: http://doi.acm.org/10.1145/139669.140382

[25] S. N. Damianakis, A. Bilas, C. Dubnicki, and E. W. Felten, "Client-Server Computing on Shrimp," *IEEE Micro*, vol. 17, no. 1, pp. 8–18, Jan. 1997. [Online]. Available: http://dx.doi.org/10.1109/40.566186

[26] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 489–502. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616493

[27] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing Network Protocols at User Level," *IEEE/ACM Trans. Netw.*, vol. 1, no. 5, pp. 554–565, Oct. 1993. [Online]. Available: http://dx.doi.org/10.1109/90.251914

[28] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-level Network Interface for Parallel and Distributed Computing," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 40–53. [Online]. Available: http://doi.acm.org/10.1145/224056.224061

[29] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 135–148. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387894

[30] L. Soares and M. Stumm, "FlexSC: Flexible System Call Scheduling with Exception-less System Calls," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924946

[31] L. Rizzo, "netmap: A novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, 2012, pp. 101–112. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rizzo

[32] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "DaRPC: Data Center RPC," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 15:1–15:13. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670994

[33] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 49–65. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[34] P. MacArthur and R. D. Russell, "A performance study to guide rdma programming decisions," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, ser. HPCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 778–785. [Online]. Available: http://dx.doi.org/10.1109/HPCC.2012.110

[35] P. Balaji, H. V. Shah, and D. Panda, "Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck," September 2004.

[36] J. Liu, J. Wu, and D. K. Panda, "High Performance RDMA-based MPI Implementation over infiniBand," *Int. J. Parallel Program.*, vol. 32, no. 3, pp. 167–198, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:IJPP.0000029272.69895.c1

[37] G. Shipman, T. Woodall, R. Graham, A. Maccabe, and P. Bridges, "Infiniband scalability in Open MPI," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006, pp. 10 pp.–.

[38] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad, "NFS over RDMA," in *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, ser. NICELI '03. New York, NY, USA: ACM, 2003, pp. 196–208. [Online]. Available: http://doi.acm.org/10.1145/944747.944753

[39] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High Performance RDMA-based Design of HDFS over InfiniBand," in *Proceedings of the International Conference on High Performance Computing, Networking,*

Storage and Analysis, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 35:1–35:35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389044

[40] M. Wasi-ur Rahman, N. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. Panda, "High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1908–1917.

[41] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda, "Memcached Design on High Performance RDMA Capable Interconnects," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 743–752. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2011.37

[42] C. Mitchell, Y. Geng, and J. Li, "Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 103–114. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535475

[43] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA Efficiently for Key-value Services," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 295–306. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626299

[44] C.-C. Tu, C.-t. Lee, and T.-c. Chiueh, "Marlin: A Memory-based Rack Area Network," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14. New York, NY, USA: ACM, 2014, pp. 125–136. [Online]. Available: http://doi.acm.org/10.1145/2658260.2658262

[45] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[46] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 3–18. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541965