

CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Department of Computer Science
Chair of Computer Architecture

Diploma Thesis

Optimizing Point-to-Point Ethernet Cluster Communication

Mirko Reinhardt

Chemnitz, February 28, 2006

Supervisor: Prof. Dr. Wolfgang Rehm

Advisor: Dipl.-Inf. Torsten Höfler

Abstract

This work covers the implementation of a raw Ethernet communication module for the Open MPI message passing library. Thereby it focuses on both the reduction of the communication latency for small messages and maximum possible compatibility. Especially the need for particular network devices, adapted network device drivers or kernel patches is avoided. The work is divided into three major parts: First, the networking subsystem of the version 2.6 Linux kernel is analyzed. Second, an Ethernet protocol family is implemented as a loadable kernel module, consisting of a basic datagram protocol (EDP), providing connection-less and unreliable datagram transport, and a streaming protocol (ESP), providing connection-oriented, sequenced and reliable byte streams. The protocols use the standard device driver interface of the Linux kernel for data transmission and reception. Their services are made available to user-space applications through the standard socket interface. Last, the existing Open MPI TCP communication module is ported atop the ESP. With bare EDP/ESP sockets a message latency of about 30 μ s could be achieved for small messages, which compared to the TCP latency of about 40 μ s is a reduction of 25 %.

Theses

Thesis I. *Open MPI and its modular component architecture are very suitable for integrating new communication protocols.*

Thesis II. *The Open MPI message passing library does not introduce much latency itself for messages sent over the network.*

Thesis III. *By eliminating the processing overhead of the TCP/IP protocol suite from the communication path the message latency can be reduced.*

Thesis IV. *It is not necessary to circumvent the Linux networking subsystem or the whole Linux kernel to achieve a significant message latency improvement.*

Thesis V. *It is not possible to use Raw Ethernet as the only communication protocol for Open MPI parallel applications. Rather a new slim and fast protocol suite based upon Ethernet has to be developed providing all the needed features.*

Thesis VI. *The need for adapted network device drivers or patched Linux kernels greatly decreases the compatibility, applicability and therefore the acceptance of a high performance protocol.*

Thesis VII. *The Linux kernel provides an open interface to easily integrate new communication protocols without the need for adapted device drivers or kernel patches.*

Contents

Chapter 1: Introduction.....	1
1.1 Motivation.....	1
1.2 Task Formulation.....	1
Chapter 2: Analysis of the Linux Networking Subsystem.....	2
2.1 The Linux Networking Subsystem.....	2
2.2 Main Networking Data Structures.....	3
2.2.1 BSD Sockets.....	3
2.2.2 The Protocol Layer's Socket Representation.....	4
2.2.3 The Protocol Interface Functions.....	9
2.2.4 Socket Buffers.....	11
2.2.5 Network Devices.....	13
2.3 Raw Packet Sockets.....	14
2.3.1 Protocol Family Registration.....	15
2.3.2 Socket Creation.....	16
2.3.3 Receiving a Data Frame.....	17
2.3.4 Requesting a Received Packet.....	22
2.3.5 Sending a Data Frame.....	23
Chapter 3: Evaluation of Existing Solutions.....	26
3.1 U-Net: A User-Level Network Interface for Parallel and Distributed Computing.....	26
3.2 M-VIA: A Modular Implementation of the Virtual Interface Architecture.....	27
3.3 Bobnet: High Performance Message Passing for Commodity Networking Components.....	28
3.4 GAMMA: The Genoa Active Message MACHine.....	28
3.5 EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing.....	29
3.6 Evaluation Summary.....	29
Chapter 4: Implementation of an Ethernet Datagram Protocol.....	31
4.1 The EDP Datagram Structure.....	32
4.2 Implementation of the Ethernet Datagram Protocol.....	33
4.2.1 Registering the Ethernet Datagram Protocol.....	33
4.2.2 The EDP Address Structure.....	33
4.2.3 Binding a Local EDP Port.....	33
4.2.4 Sending an EDP Datagram.....	35
4.2.5 Receiving Incoming EDP Datagrams.....	35
4.2.6 Additional Protocol Interface Functions Implemented by EDP.....	36
4.3 EDP Micro-Benchmarks.....	36
4.4 Zero-Copy Data Transmission.....	38
4.4.1 A Simple Zero-Copy Sending Approach.....	39
4.4.2 Zero-Copy Sending Benchmarks.....	40
4.4.3 Problems of the Zero-Copy Sending Approach.....	42
4.5 Conclusion.....	42
Chapter 5: Implementation of an Ethernet Streaming Protocol.....	45
5.1 The ESP Packet Structure.....	45
5.2 Implementation of the Ethernet Streaming Protocol.....	46
5.2.1 Registering the ESP Protocol.....	46

5.2.2 ESP Socket States.....	46
5.2.3 Socket Management.....	47
5.2.4 Initiating a Connection.....	48
5.2.5 Accepting a Connection.....	49
5.2.6 Suspending the Current Process.....	50
5.2.7 Sending User Data.....	51
5.2.8 Managing Retransmissions.....	51
5.2.9 Receiving ESP Data Packets.....	52
5.2.10 Processing Received Packets.....	53
5.2.11 Acknowledging Received Packets.....	54
5.3 ESP Micro-Benchmarks.....	55
5.3.1 Message Latency and Throughput.....	55
5.3.2 Influence of Acknowledgement Processing.....	57
5.3.3 Average Throughput, Influence of the Socket Buffer Sizes.....	58
Chapter 6: Open MPI v1.0.....	59
6.1 The Modular Component Architecture.....	59
6.2 The TCP BTL Component.....	60
6.2.1 The Lifecycle of the TCP BTL Component.....	60
6.2.2 Managing Remote Processes.....	61
6.2.3 Data Transmission.....	61
6.3 Implementation of an Ethernet BTL Component.....	62
6.4 Micro-Benchmarks.....	63
6.4.1 Benchmarks of the Pallas MPI Benchmark Suite.....	63
6.4.2 Communication Module Profiling.....	64
Chapter 7: Summary and Conclusion.....	lxvi
Appendix.....	lxvii
A.1. References.....	lxvii
A.2. List of Figures.....	lxix
A.3. List of Abbreviations.....	lxx
A.4. The Cluster Test Systems.....	lxxi

Chapter 1: Introduction

1.1 Motivation

Ethernet is currently the ubiquitous technology used in wide-area networks. With the new 10 GB Ethernet standard it becomes also interesting for system-area networks, like cluster environments, that are currently often interconnected by other high speed communication means. Additionally, as Ethernet adapters are often contained in off-the-shelf computer systems, it is an inexpensive alternative in low-cost commodity cluster systems. Ethernet, however, is mostly used in combination with additional protocols, e.g. the TCP/IP protocol family, which may introduce an additional processing overhead and provide features that are not necessary in cluster environments. Therefore the possibility of using raw Ethernet for cluster communication shall be analyzed and an optimized communication means based upon Ethernet shall be developed.

1.2 Task Formulation

The objective of this Diploma Thesis is an implementation of the Message Passing Interface (MPI) Standard optimized for Ethernet networks. The work will be based upon a current version of Open MPI, that shall be extended and optimized by the circumvention of the TCP/IP protocol stack. Thereby it is most important, that the coding standards and concepts prescribed by Open MPI are observed. A zero-copy solution of the problem shall be discussed and an explanation shall be given, if it cannot be realized. Additionally this work shall contain an analysis of the networking subsystem of the version 2.6. Linux kernel and a documentation of all used kernel functions.

The existing implementation of the Open MPI TCP BTL component can be used as an example and for performance comparison.

Portability is an important aim of the work. Therefore only few kernel functions should be used and kernel extensions should be implemented as a kernel module. The result shall be executable on both the *ia32* and *em64t* hardware architectures. It should be independent from particular hardware or network devices and shall be able to support multiple interfaces in a single system.

Chapter 2: Analysis of the Linux Networking Subsystem

2.1 The Linux Networking Subsystem

Linux, being a versatile multi-purpose operating system, incorporates a very flexible networking subsystem. Inspired from the original Berkeley Unix implementation it supports many different network architectures and provides various tools for system administrators to set up routers, gateways or firewalls. As shown in *Figure 2.1.1* it is organized into layers, that partly represent the ISO/OSI network model. Adjacent layers are interconnected by well-defined interfaces.

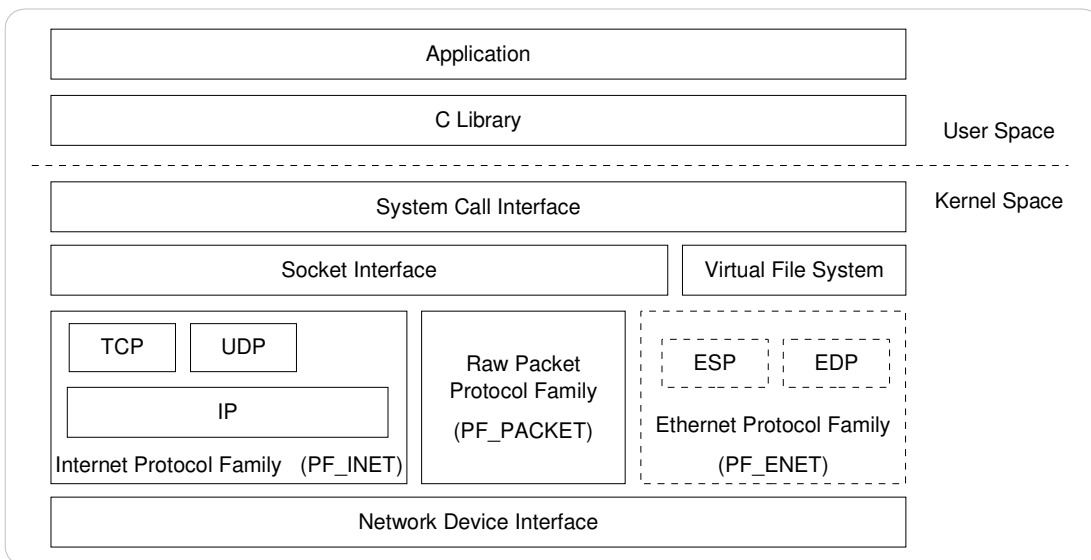


Figure 2.1.1: The Linux Networking Subsystem

The Linux networking subsystem supports many different communication protocols, with related ones grouped to protocol families. Additionally, it provides the possibility to register new protocol families with the socket and network device interfaces. Using the technique of kernel modules they can even be added to or removed from a running system. To user-space applications protocol families are represented by an integer identifier.

User space applications can utilize the services provided by the networking subsystem using sockets, i.e. data structures representing a virtual communication endpoint, and related system calls. Following the Unix philosophy, each socket is closely related to a file of the *sockfs* special file system. To an application sockets are represented by the integer descriptor of that file.

Whenever data is sent or received, it must traverse the layers of the networking subsystem and the involved protocols. The kernel thereby avoids the copying of data as much as possible. A special data structure is used containing a memory buffer, that provides enough space for the data as well as additional control information of any communication protocol involved, and a descriptor needed to manage the buffer efficiently.

2.2 Main Networking Data Structures

2.2.1 BSD Sockets

Generally speaking, any operating system has to define an entry point for user mode applications, an Application Programming Interface (API), to each service it provides. Linux uses BSD *sockets* to provide access to the kernel's networking subsystem. They were introduced in Berkeley's Unix 4.1cBSD and are available in almost all Unix-like operating systems. Conforming to the Unix philosophy, Linux implements BSD sockets as files that belong to the *sockfs* special file system. Their attributes are stored in a `struct socket` data structure, which is connected to the file's inode.

The most important fields of the BSD socket object are:

`short type`

Denotes the type of the socket and can be one of the constants

<code>SOCK_DGRAM</code>	unreliable, fixed maximum length datagram transport
<code>SOCK_RDM</code>	reliable datagram transport that does not guarantee ordering
<code>SOCK_SEQPACKET</code>	sequenced, reliable, connection-oriented datagram transport
<code>SOCK_STREAM</code>	sequenced, reliable, connection-oriented byte streams
<code>SOCK_RAW</code>	raw network protocol access

The still present `SOCK_PACKET` socket type is obsolete and should not be used anymore. The `PF_PACKET` protocol family should be used instead, which is described in detail in section 2.3.

`struct file *file`

Points to the associated file object of the *sockfs* special file system.

`socket_state state`

The protocol can use this field to store and check the connection state of the socket. It can be one of the constants

<code>SS_FREE</code>	the socket is not allocated,
<code>SS_UNCONNECTED</code>	the socket is not connected,
<code>SS_CONNECTING</code>	the socket is in process of connecting,
<code>SS_CONNECTED</code>	the socket is connected,
<code>SS_DISCONNECTING</code>	the socket is in process of disconnecting

`struct proto_ops *ops`

This field points to the protocol's interface functions that shall be used when an application issues a socket call. Generally speaking, they are the kernel's protocol dependent counterparts of the user-level socket functions. The single functions are described in detail in section 2.2.3.

`struct sock *sk`

Points to the protocol layer's socket representation data structure.

The BSD socket object mainly serves as a connection between the *sockfs* special file system and the networking subsystem and as an unified interface to the underlying protocol. Its representation is mainly managed by the kernel itself, i.e. the protocol family is not responsible for the creation, initialization or destruction of BSD socket objects. But it is notified of those events to provide its own socket representation and protocol interface

functions at socket creation time and take whatever action is necessary to cleanup all resources (especially the provided socket representation) at socket destruction time. Furthermore, a `struct socket` object is passed as parameter to virtually all protocol interface functions to indicate the socket to work with.

2.2.2 The Protocol Layer's Socket Representation

In the protocol layer sockets are represented by a `struct sock` data structure which is connected to the BSD socket object. The protocol itself is responsible to create and initialize this structure upon socket creation and to establish the link to the BSD socket. Additionally it is possible to connect a protocol-private data structure to the `struct sock` object using one of the two ways shown in *Figure 2.2.1*. First, the private data structure can be allocated separate from the socket object and linked to it using a pointer. But it is more efficient to allocate both the socket object and the private data structure together from a protocol-specific memory slab cache [1], especially when sockets are created and destroyed in rapid succession.

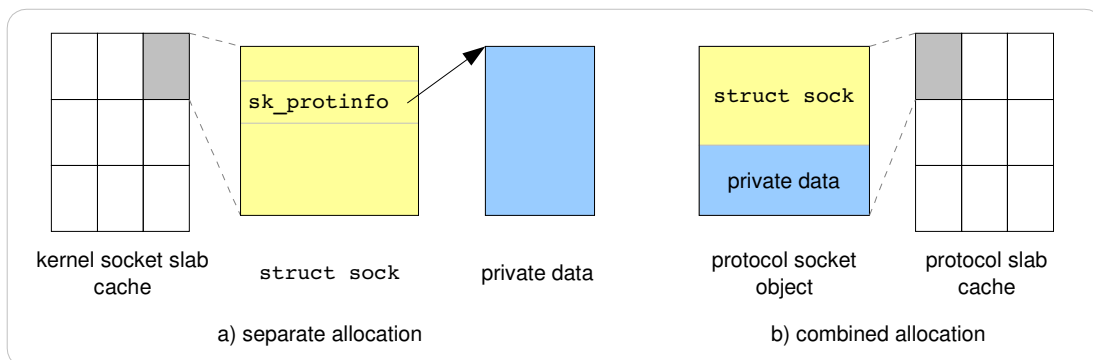


Figure 2.2.1: The Protocol Layer Socket and Private Data Structures

The protocol layer socket object is allocated using the `sk_alloc()` function. The first two parameters this function expects are the protocol family's identifier and the allocation priority¹. While in user contexts usually `GFP_KERNEL` is used for the priority, it is essential to provide `GFP_ATOMIC` when sockets shall be allocated in interrupt context or when locks are held. Otherwise the allocation function might block when no free memory is available.

The following parameters differ between kernels prior to version 2.6.12 and more recent ones. The third parameter of the older version of the allocation function is an integer that specifies whether the new socket object should be filled with zero bytes. If a value of 1 is provided, the function assumes the new socket to be of the size of `struct sock`. A value greater than one denotes the size of the object to be zero-filled. This is especially useful when the socket object is allocated together with a protocol-private data structure from the slab cache specified using the last parameter of the allocation function. If `NULL` is specified here the kernel uses an internal cache to obtain the socket structure from.

The more recent version of the allocation function expects a data structure of type `struct proto` as third parameter, which contains the name of the protocol (for the `procfs` special file system), the size of the socket objects to allocate and possibly fill with zero bytes and a pointer to the slab cache to use. The protocol structure should be

¹ sometimes also called `gfp_mask` for `get_free_pages()` mask

registered with the kernel upon protocol initialization using the `proto_register()` function that can also create the slab cache for the protocol's socket objects and enter its address into the structure, if requested. Upon protocol destruction the structure can be unregistered using `proto_unregister()` that also destroys a possibly associated slab cache. The last parameter the socket allocation function takes is a flag that indicates whether to zero-fill the allocated socket object.

After the socket data structure has been allocated it is managed using reference counting. Newly allocated sockets have a reference count of one. Whenever a socket is used beyond the scope of a single function, e.g. it is inserted into a list or used in combination with a timer, `sock_hold()` should be called to increment its reference count. Contrary to that `sock_put()` shall be called when the long-term use of the socket ends to decrease its reference count. This function will automatically invoke `sk_free()` to release the socket object when its reference count drops to zero, which in turn calls the destructor `sk->sk_destruct()` associated with the socket before its descriptor (or the whole object, depending on the allocation) is finally returned to the slab cache it was allocated from.

The most important fields and function pointers of the `struct sock` object are:

```
unsigned short sk_family
```

The protocol family this socket belongs to.

```
volatile unsigned char sk_state
```

This field is used by stateful protocols, like TCP, to manage the state of the socket and the connection in a more fine-grained way than the BSD socket state allows. Although it is basically possible to define and use any private state constant needed, some of the predefined kernel functions either set this field to one of the TCP states, like `sock_init_data()`, or expect one of the TCP states in it, like `datagram_poll()` or `wait_for_packet()`.

```
struct hlist_node sk_node
```

This field is used by the kernel function `sk_add_node()` to add the socket to a list or hashtable. The function increments the socket's reference count to prevent it from being released while it is part of the list, which would tear the list apart. The functions `sk_hashed()` and `sk_unhashed()` can be used to check, whether a particular socket is currently in a list, while `sk_del_node()` removes the socket from the list, decrementing its reference count. Since it leaves the socket's `sk_node` field at an undefined value, the mentioned list checking functions cannot be applied to the socket afterwards. Therefore it is often better to use `sk_del_node_init()` to remove the socket from a list, which also re-initializes the socket's `sk_node` field properly.

```
struct hlist_node sk_bind_node
```

The functions `sk_add_bind_node()` can be used to add the socket to a secondary list using this field, while `sk_del_bind_node()` removes the socket from the list. Unlike the previously mentioned functions these two do no reference counting.

The macros `sk_for_each()` or `sk_for_each_bound()` can be used to iterate over immutable lists of sockets according to the used list node field, while

`sk_for_each_safe()` can be used when sockets shall be removed from the list during the iteration.

`atomic_t sk_refcnt`

This field indicates the socket's reference count. It is virtually never accessed directly. Rather `sock_hold()` should be called to increment the reference count ("to grab a reference"), while `sock_put()` decrements it and calls `sk_free()` to release the socket if the reference count drops to zero.

`socket_lock_t sk_lock`

This lock can be used to serialize the access to the socket. It contains a spinlock to synchronize between user context and interrupt processing and a simple semaphore to synchronize amongst multiple users. In the protocol interface functions `lock_sock()` and `release_sock()` can be called to acquire or release the semaphore, respectively. The spinlock is held to protect these actions, but afterwards it is released immediately. Interrupt handlers, which are used to receive data asynchronously, can use the `bh_lock_sock()` and `bh_release_sock()` macros to acquire or release the spinlock. This procedure still allows the interrupt handler to be executed concurrently with user context processing, as it is needed to achieve maximum performance and minimum data loss. Therefore interrupt handlers of stateful sockets can check whether the semaphore is currently locked using the `sock_owned_by_user()` macro. They can then take precautions to prevent the socket's state from changing unexpectedly by the user space process.

`int sk_rcvbuf`

`int sk_sndbuf`

These fields indicate how much of the system's main memory a socket may consume for its receive or send queues, respectively. User-space processes can read or set these values using the `getsockopt()` or `setsockopt()` functions providing the `SOL_SOCKET` socket level constant and the `SO_RCVBUF` or `SO_SNDBUF` option name. The system-wide default and maximum of these values can be adjusted by a privileged user with the `sysctl` command providing the `net.core.rmem_default`, `net.core.wmem_default`, `net.core.rmem_max` or `net.core.wmem_max` option names.

`atomic_t sk_rmem_alloc`

`atomic_t sk_wmem_alloc`

These fields indicate how much of the system's main memory a socket currently uses for its receive or send queues. Socket buffers received or about to be sent can be charged to the socket's memory using the `skb_set_owner_r()` or `skb_set_owner_w()` functions. They additionally set the destructor of the socket buffers to the `sock_rfree()` or `sock_wfree()` functions which return the buffer's memory to the socket upon the buffer's destruction.

Although the socket's upper memory bounds are managed by the kernel itself, it is the protocol's task to ensure that they are not exceeded. Whenever received data is handed to the protocol, it must check whether there is enough receive memory left at the target socket before the data can be enqueued for further processing. If a socket is out of

receive memory, the most common action is to simply discard the received data. When data is handed to the kernel for transmission using the `dev_queue_xmit()` function, it will first be queued in a device specific queue. The data buffer will not be released until the physical transmission has completed. Therefore, the protocol must check whether enough write space is available at the socket whenever a new buffer has to be allocated for sending. If a socket is out of send memory, it should block the sending process or return the `EAGAIN` error code, if the socket is switched to non-blocking mode. If a socket's memory limits are not observed, it could render the whole system unusable by allocating all of the available memory.

```
struct sk_buff_head sk_receive_queue
```

Any data received by the network device layer will be handed to the protocol in the form of socket buffers, represented by `struct sk_buff` data structures. It can then be appended to a socket's receive queue by the data reception interrupt handler. Data reception socket calls, e.g. `recv()`, performed by an user space process will finally end up in the corresponding socket's `recvmsg()` interface function being executed, which removes pending data from the receive queue and hands it to the user.

```
struct sk_buff_head sk_write_queue
```

```
struct sk_buff *sk_send_head
```

```
int sk_wmem_queued
```

Connection-oriented sockets use these fields (amongst others) to organize the data window and the retransmission of lost data segments. Every data buffer is enqueued to the write queue before it is sent. The send head points to the data buffer next to send. Thereby buffers before the send head contain data already sent but not yet acknowledged, while buffers after the send head still have to be transmitted. The `sk_wmem_queued` field indicates the memory allocated by the retransmission queue.

```
unsigned short sk_ack_backlog
```

```
unsigned short sk_max_ack_backlog
```

These fields can be used by connection-oriented sockets to manage the pending connection requests. The maximum of pending connection requests for a particular socket is specified by user-space processes when the socket is switched to listening mode using the `listen()` socket call. The system-wide maximum for this value can be adjusted by a privileged user with the `sysctl` command providing the `net.core.somaxconn` option name. While the kernel ensures that the user doesn't provide a value bigger than the system-wide maximum to the `listen()` socket call, it is up to the protocol to store this value into the socket's `sk_max_ack_backlog` field. The protocol is also responsible to accept only as many connection requests on a socket as its maximum allows. The functions `sk_acceptq_added()` and `sk_acceptq_removed()` can be used to count the pending requests, while `sk_acceptq_is_full()` can be used to test, whether the maximum has been reached.

```
int sk_err
```

This field can be used for socket error reporting. Unlike the protocol interface functions, which return negativ error codes, a positive value is expected here. After the error

code has been set, the `sk_error_report()` function of the socket should be called to wake up any processes sleeping in a blocking socket call. The protocol interface function the process was waiting in can then take the value of the `sk_err` field and return it to the user.

```
struct socket *sk_socket
```

The associated BSD socket object can be accessed with this field.

```
void *sk_protinfo
```

This pointer provides a means for a protocol to associate a private data structure with the protocol layer socket representation. The protocol could allocate the private data structure and associate it with the socket upon its creation. It is essential in this case to free the allocated memory in the socket's destructor to avoid memory leakage.

Another, more efficient way is to allocate both the `struct sock` and the private data structure together from a slab cache as described before.

```
void (*sk_state_change)(struct sock *sk)
```

```
void (*sk_data_ready)(struct sock *sk, int bytes)
```

```
void (*sk_write_space)(struct sock *sk)
```

```
void (*sk_error_report)(struct sock *sk)
```

These functions associated with a socket can be used to indicate an event to processes working with the socket, e.g. whenever the socket's state changes, data becomes ready to be received, there is free write memory available to proceed with data transmission or an error condition occurred. The function `sock_init_data()`, which can be used to conveniently initialize the socket data structure, sets these function pointers to default functions provided by the kernel. The default functions, when invoked, will wake up any processes sleeping in a blocking socket call and send the appropriate signals to processes working with sockets in asynchronous mode.

```
int (*sk_backlog_rcv)(struct sock *sk, struct sk_buff *skb)
```

Stateful sockets must be protected against unexpected state changes. The socket's lock can be used to fulfil part of the task. But it does not prevent the data reception interrupt handler from being executed while a socket call is in progress. Therefore the interrupt handler should use the `sock_owned_by_user()` macro to determine whether the socket is currently in the user's hands. If so, it can use the `sk_add_backlog()` function to temporarily queue the received data buffer, which could cause a state change, into the socket's `sk_backlog` queue. When the user context processing calls `release_sock()` to release the socket's lock, the backlog queue is processed and the `sk_backlog_rcv()` function is called for every data buffer kept therein to perform the actual data receiving.

```
void (*sk_destruct)(struct sock *sk)
```

This function represents the socket's destructor. It is called by the `sock_put()` function when the socket's reference count drops to zero. When a separate private data structure has been allocated and associated with the socket, it should be freed here. It is also the place for any other cleanup before the socket is finally destroyed by the kernel.

2.2.3 The Protocol Interface Functions

The standard C library translates all socket calls, like `send()`, `recv()` or `connect()`, into the `sys_socketcall()` system call, which serves as a multiplexer and invokes the appropriate kernel socket function. Those, after having determined the appropriate BSD socket object, forward the call to the corresponding protocol interface function contained in the BSD socket's `*ops` field, which is of type `struct proto_ops` and contains the following fields and function pointers:

```
int family
```

Contains the identifier of the protocol family these operations belong to.

```
struct module *owner
```

Specifies the module the operations are located in. This field is used by the kernel to perform module reference counting.

```
int (*bind) (struct socket *sock, struct sockaddr *myaddr,
            int sockaddr_len)
```

This function handles the `bind()` socket call by assigning the requested local address to the socket.

```
int (*listen) (struct socket *sock, int len)
```

```
int (*accept) (struct socket *sock, struct socket *newsock,
              int flags)
```

```
int (*connect) (struct socket *sock, struct sockaddr *vaddr,
               int sockaddr_len, int flags)
```

The user-space `listen()`, `accept()` and `connect()` socket calls are handled by these functions. Connection-oriented protocols, like TCP, can use `listen()` to switch a socket to listening mode and adjust the maximum length of the internal queue of pending connection requests. Afterwards the `accept()` function can be invoked to fetch a connection request from the queue. The `connect()` function may be used by connection-oriented protocols to initiate a complex connecting procedure, while connection-less protocols, like UDP, might use it to just store the remote address for subsequent data reception or transmission calls.

Because the `connect()` and `accept()` socket calls don't offer a parameter to provide any flags, the flags handed to these two functions are taken from the file associated to the BSD socket object. These can be adjusted from user-space applications using the `fcntl()` function. Especially for non-blocking operation the `O_NONBLOCK` option is used here rather than the `MSG_DONTWAIT` flag given to the data reception or transmission functions.

```
int (*sendmsg) (struct kiocb *iocb, struct socket *sock,
               struct msghdr *m, size_t total_len)
```

```
int (*recvmsg) (struct kiocb *iocb, struct socket *sock,
               struct msghdr *m, size_t total_len, int flags)
```

These two functions handle nearly all data sending and receiving performed using a socket. This includes the `read()`, `readv()`, `recv()`, `recvfrom()` and `recvmsg()` file and socket calls and their writing counterparts `write()`, `writew()`, `send()`, `sendto()` and `sendmsg()`.

The i/o control block given as the first parameter is normally used to manage asynchronous i/o operations, while it is completely unused e.g. by the internet or raw packet protocol families.

The second parameter of type `struct msghdr` describes where the data and remote addressing information shall be obtained from or copied to. The `msg_name` field of this structure points to the remote address buffer of the size indicated by the `msg_namelen` field. The structure's `msg_iov` field points to an array of i/o vectors, i.e. address-length tuples, describing the position and size of one or more data buffers the user space application provided. Its `msg_iovlen` field indicates the amount of i/o vectors contained in that array. The kernel offers some convenience functions for working with i/o vectors, like `memcpy_fromiovec()`, `memcpy_toiovec()` or `skb_copy_datagram_iovec()`, which will modify the given vector on each invocation to indicate the amount of data copied. This way they can be called multiple times in succession e.g. to fragment or re-assemble a larger data block.

```
unsigned int (*poll) (struct file *file, struct socket *sock,
    struct poll_table_struct *wait)
```

The `select()` and the related `poll()` socket calls are handled by this function. It returns a mask of flags that indicate the current socket state, e.g. whether it is readable, writable or if connections or errors are pending.

```
int (*shutdown) (struct socket *sock, int flags)
```

This is the protocol's `shutdown()` handler, which is used to close one endpoint of a full-duplex connection for sending and/or receiving.

```
int (*release) (struct socket *sock)
```

The release function of a BSD socket is called when the application wants to `close()` the socket. It should release the protocol's socket representation, flush queues of pending data and take any other action necessary to cleanup the resources occupied by this socket.

```
int (*socketpair) (struct socket *sock1, struct socket *sock2)
```

The `socketpair()` call is used to create two identical initially connected sockets. A forking application could do so and inherit the socket handles to install a communication channel between the parent and child processes.

```
int (*getname) (struct socket *sock, struct sockaddr *addr,
    int *sockaddr_len, int peer)
```

This function handles both the `getsockname()` and `getpeername()` socket calls used by applications to determine the local address of a socket or the remote address of a connected peer. The `peer` parameter is used to distinguish between them – it is zero in the former and non-zero in the latter case.

```
int (*ioctl) (struct socket *sock, unsigned int cmd,
    unsigned long arg)
```

This is the protocol's handler of the `ioctl()` socket call. There's mostly only a few `ioctl` commands that are handled by the protocol layer, e.g. `SIOCOUTQ` or `SIOCINQ` to determine the amount of data waiting to be sent or received, respectively. Others should be forwarded to the network device layer using the `dev_ioctl()` function.

```
int (*setsockopt) (struct socket *sock, int level, int optname,
    char __user *optval, int optlen)
```

```
int (*getsockopt) (struct socket *sock, int level, int optname,
    char __user *optval, int __user *optlen)
```

These functions handle the `setsockopt()` and `getsockopt()` socket calls, which set or get socket options, respectively. While the `optname` parameter specifies the particular option to access, the `level` parameter defines the layer of the networking subsystem where the parameter should be adjusted or read. The `SOL_SOCKET` constant for instance is used to access overall BSD socket options, which are completely managed by the kernel. If one of the protocol-dependent constants, like `SOL_IP`, `SOL_UDP` or `SOL_TCP`, is used these protocol-specific functions will be called.

```
int (*mmap) (struct file *file, struct socket *sock,
    struct vm_area_struct * vma)
```

The protocol's `mmap()` handler, which implements file-like memory mapping. Currently only raw packet sockets support this method.

```
ssize_t (*sendpage) (struct socket *sock, struct page *page,
    int offset, size_t size, int flags)
```

This function is called during the `sendfile()` system call, which is used to copy data between two open file descriptors within the kernel and thus without the time consuming copying operations to and from user space.

Every protocol has to provide an interface function structure filled with pointers to its own implementations when a new socket is created. The kernel defines default functions, like `sock_no_accept()` or `sock_no_mmap()`, that return an appropriate error code or map the call to another function, like `sock_no_sendpage()`, that can be used, if the protocol doesn't support a particular interface function.

It is common to nearly all interface functions that they return a negative error code upon failure or zero or an appropriate positive value on success, like the sending and receiving functions, which return the number of bytes transferred. An exception is the `poll()` handler, which cannot return a negative error code.

2.2.4 Socket Buffers

When data is transmitted or received it usually traverses the layered network protocol architecture top-down or bottom-up, respectively. Upon transmission every involved protocol will add its own control information to the data, i.e. a specific header might be prepended as well as a trailer might be appended. Upon the reception of a data packet each protocol must remove its own header and trailer to reveal the information needed by the next higher level protocol and eventually the original user data payload. The Linux kernel uses a special data structure called *socket buffer* (`skb`) to perform these tasks efficiently. A single socket buffer consists of two parts: a descriptor of type `struct sk_buff` and an associated data buffer.

Socket buffers can be allocated using the function `alloc_skb()` directly. But the Linux kernel also offers some convenience functions like `sock_wmalloc()` or `sock_alloc_send_skb()`, that additionally check if the socket has enough buffer space left, charge the allocated socket buffer to the socket's memory or even block the

current process, if there is no memory left. Besides the needed buffer size these allocation functions again expect an allocation priority parameter, like the allocation function for socket objects.

The result of the allocation functions is a pointer to a newly allocated socket buffer's descriptor with a usage count of one or `NULL`, if the allocation failed. And it is this pointer that is passed to each of the involved protocols, that add or remove their control information to or from the data as needed. While this practice avoids the copying of data across the layers it introduces a new risk. It is possible that a single socket buffer is handed to multiple threads of processing simultaneously, especially when network analysers are peeking at the traffic. Therefore the consumers of a socket buffer must take precautions not to interfere with each other. If one of them wants to change any of the descriptor's fields, it must create a private copy of it using `skb_clone()`. This holds even true if the socket buffer shall only be enqueued, because it's descriptor does only provide a single attachment point and can thus only be in a single queue at a time. If one of a socket buffer's consumers even wants to change the data contained in the packet it must create a copy of both the descriptor and the data buffer using `skb_copy()`.

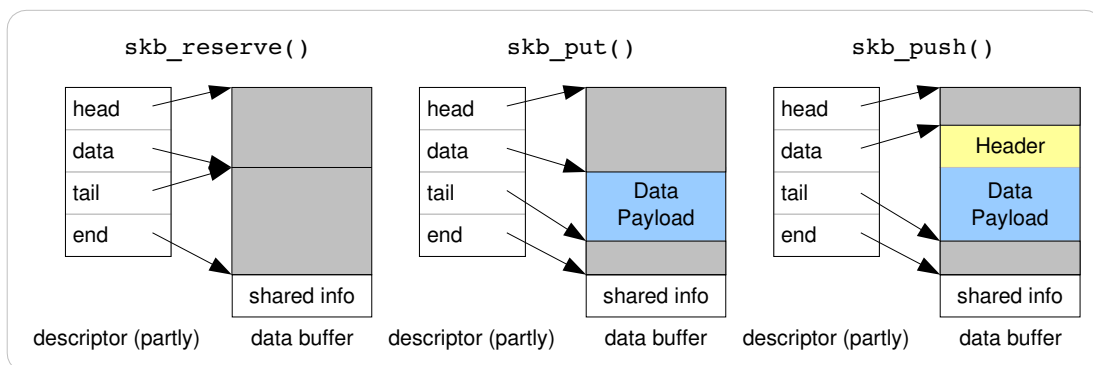


Figure 2.2.2: Linux Socket Buffer Pointer Operations

After the allocation the socket buffers are managed using reference counting. The descriptor's usage count can be incremented using the `skb_get()` function. To release a socket buffer the `kfree_skb()` function can be called. It will decrease the descriptor's usage count by one and finally free the descriptor when it drops to zero. The data buffer contains a separate reference count in its shared info data area, because it can be referenced by multiple cloned descriptors at the same time. It is automatically managed by the socket buffer cloning, copying and releasing functions and `kfree_skb()` will additionally release the data buffer when it drops to zero.

The socket buffer's descriptor contains the following important fields:

```
struct sock *sk
```

This field indicates the socket owning the buffer. It is especially set by the `skb_set_owner_r()` and `skb_set_owner_w()` functions and used by the associated socket buffer destructors `sock_wfree()` and `sock_rfree()` to determine the socket to return the memory consumed by the socket buffer to.

```
struct net_device *dev
```

For socket buffers of received data this field indicates the device through which the data entered the system. Before a socket buffer can be handed to the device layer for

transmission using the `dev_queue_xmit()` function this field has to be set to the desired output device.

```
char cb[40]
```

In current version 2.6 Linux kernels the socket buffer's control block provides space for 40 bytes of protocol-private information. The socket buffer's descriptor must be cloned when the information shall be preserved over several protocol layers to prevent other consumers of the socket buffer from changing the private data.

```
unsigned int len
```

This field indicates the amount of data contained in the socket buffer. It is updated by any of the functions changing the descriptor's data pointers.

```
unsigned int truesize
```

This field indicates the true memory consumption of the socket buffer. Due to alignment padding its value is mostly bigger than that of the `len` field.

```
unsigned char *head, *data, *tail, *end
```

While the `head` and `end` pointers mark the boundaries of the data buffer and the additional header pointers may be used by the protocols to access the link layer (`mac`), network layer (`nh`) or transport layer (`h`) header, the `data` and `tail` pointers are used to prepend, append or remove protocol control information to or from the data.

The `head`, `data` and `tail` pointers of a newly allocated socket buffer initially point to the beginning of its data buffer. But the kernel provides several functions to manipulate a socket buffer's pointers. The effect of `skb_reserve()`, `skb_put()` and `skb_push()` on them, as used normally when preparing a socket buffer for transmission, is depicted in *Figure 2.2.2*. To "remove" a protocol header from a received socket buffer the `skb_pull()` function can be used, which advances the `data` pointer towards the end of the data buffer. Although these functions do only adjust the data pointers of a socket buffer's descriptor, all but `skb_reserve()` return a pointer to use e.g. with `memcpy()` to fill the actual data or control information into its data buffer to the appropriate position.

The socket buffers considered so far are also called *linear*, because the whole data is kept in one consecutive data buffer. Additionally, the shared info data structure kept at the end of the data buffer provides the possibility to dynamically extend the data space without the need for re-allocation or data copying. For that purpose a list of additional socket buffers or a set of auxiliary memory pages can be appended to an existing socket buffer. The kernel offers a set of functions to manipulate such *non-linear* socket buffers that take their extensions into account, like `pskb_pull()` or `pskb_trim()`.

2.2.5 Network Devices

The Linux kernel uses descriptors of type `struct net_device` to manage installed network devices.

```
char name[IFNAMSIZ]
```

This field holds the name of the interface, e.g. "eth0". To determine the descriptor of a device by its name the kernel function `dev_get_by_name()` can be used.

```
int ifindex
```

The interface index uniquely identifies a network device. To determine the descriptor of a device by its index the kernel function `dev_get_by_index()` can be used. User space applications can find out the index of a network device using `ioctl()` on an open socket providing the `SIOCGIFINDEX` option or by calling the convenience function `if_nametoindex()`.

```
unsigned mtu
```

This field indicates the maximum transfer unit (MTU) of the device. User space applications can determine the MTU of a network device using `ioctl()` on an open socket providing the `SIOCGIFMTU` option.

```
unsigned short hard_header_len
```

```
unsigned char dev_addr[MAX_ADDR_LEN]
```

```
unsigned char addr_len
```

Information about the hardware address and header length can be obtained from these fields. User space applications can find out the hardware address of a network device using `ioctl()` on an open socket providing the `SIOCGIFHWADDR` option.

```
int (*hard_header) (struct sk_buff *skb, struct net_device *dev,
    unsigned short type, void *daddr, void *saddr, unsigned len)
```

This function of a network device driver can be used to fill in the hardware header into a socket buffer being prepared for transmission. For that purpose it will move the buffer's `data` pointer towards its head by the size needed for the header, e.g. using `skb_push()` as depicted in *Figure 2.2.2*, before copying the needed information into it. For the type parameter the desired packet type has to be provided in host byte order. For the source address pointer `NULL` can be given to enter the hardware address of the provided device, while the destination address is left empty if `NULL` is provided here. The function will return a negative value to indicate errors.

The drivers of Ethernet devices will usually set this field to the `eth_header()` function.

```
int (*hard_header_parse) (struct sk_buff *skb,
    unsigned char *haddr)
```

This function of a network device driver parses the hardware header of a socket buffer pointed to by the buffer's `mac` field and copies the packet's source address into the provided buffer. Its return value will be the byte length of the copied address.

The drivers of Ethernet devices will usually set this field to the `eth_header_parse()` function.

2.3 Raw Packet Sockets

Raw packet sockets provide access to the basic link level protocol layer. With their help an user-space application running under a privileged account is able to send data packets or frames of a link layer protocol, e.g. Ethernet. Packet sockets can be used in two different ways: In raw mode the application must provide complete packets or frames including the link level protocol header to the sending socket functions and the receiving functions will return complete packets, too. In datagram mode the addressing information

of the packet is provided and returned separate from the content using the `struct sockaddr_ll` link layer socket address structure.

Packet sockets and their operations are implemented in version 2.6 Linux kernels using the `PF_PACKET` protocol family.

2.3.1 Protocol Family Registration

To install a protocol family it is necessary to register it with both the socket and network device layers, because the socket layer needs to know that and how sockets for the new protocol family can be created and the device layer needs to know where to deliver received data to.

The registration to the socket layer is performed in the initialization function of the packet protocol family called upon module loading or system startup time, if the protocol family is compiled statically into the kernel. For that purpose the `sock_register()` function is called with a pointer to a `struct net_proto_family` as its only parameter. This data structure contains the numeric identifier of the protocol family (`PF_PACKET`) and a pointer to the function `packet_create()` which the kernel shall call whenever new packet sockets have to be created. The initialization function of the protocol family additionally installs a network device notifier function, that is called whenever a network device changes its state, e.g. goes up or down, and handler functions for calls by the `procfs` special file system to make the protocol's internal values accessible through the virtual file `/proc/net/packet`.

The kernel uses two separate data structures to manage data reception hooks: a list of packet handlers that are called for every packet received or about to be sent and a hashtable of handlers that are called for received packets of a particular type only. A data reception hook can be registered with the network device layer using the `dev_add_pack()` function, which expects a pointer to a `struct packet_type` as its only parameter. This structure contains a pointer to the function that shall be called by the kernel whenever relevant data frames are received, i.e. the `packet_rcv()` function for raw packet sockets, and some fields describing which data frames the device layer shall deliver to the protocol. The `type` field contains the numerical type of packets the protocol family wants to receive in network byte order. If the `ETH_P_ALL` special constant is specified here, the protocol's reception hook will be inserted into the list of handlers for every data packet and its data reception function will be called for every data frame received or about to be sent. Otherwise the hook is inserted into the hashtable of handlers and the kernel will use the `protocol` field of received socket buffers to determine to which of the handlers to deliver the frame. The value of this field is usually taken from the link level protocol's packet type field of the received data frame. The `dev` field of the packet type structure contains a pointer to the descriptor of the network device to receive packets from or `NULL`, if packets from all devices should be received by the protocol. Finally, the field `af_packet_priv` provides a means to associate protocol-private data to the data reception hook. The whole packet type structure used to register the protocol family will be provided to the reception function in addition to the socket buffer containing the received data and the descriptor of the input device whenever

relevant data has been received. This way also the protocol-private data becomes accessible again.

Every packet socket registers a new data reception hook for the packet type requested by the user. The actual registration can take place at different points, e.g. during socket creation, if the wanted packet type is given by the application using the `protocol` parameter of the `socket()` function, or when the socket is bound. The private data field of the `struct packet_type` is used by the packet protocol family to store a pointer to the target socket so it can easily be retrieved whenever data is received.

2.3.2 Socket Creation

User level applications can invoke the `socket()` function to create a new network socket of a particular type and for a particular protocol family and protocol. The standard C library translates all socket calls into the `sys_socketcall()` system call, which serves as a multiplexer and invokes the appropriate kernel socket function. In the current case `sys_socket()` is called.

First, `sys_socket()` invokes the `sock_create()` function to allocate a new BSD socket object. This function in turn starts checking the protocol family and socket type parameters provided by the user space application. The numerical protocol family identifier must thereby lie between zero and the value of the `NPROTO` constant, which is 32 in current 2.6 version kernels. This somewhat limits the ability to insert new protocol families into the kernel without modifying it, although there are still a few numbers unused in this range.

If no protocol family has registered for the provided identifier yet, the function next tries to load in the module that provides it. The module is searched by the protocol alias that can be specified in the module sources using the `MODULE_ALIAS_NETPROTO()` macro, e.g. with the `PF_PACKET` constant for raw packet sockets.

Afterwards `sock_create()` calls the `sock_alloc()` function to allocate a new BSD socket object. There a combined inode – socket data structure is allocated from the inode slab cache of the `sockfs` special file system similarly to the combined allocation of protocol layer socket objects described in section 2.2.2. Afterwards the inode's `sock` flag is set to indicate that it belongs to a socket object.

The `sock_create()` function now invokes the protocol family's socket construction function registered with the socket layer providing the new BSD socket object and the requested protocol as parameters.

For raw packet sockets this is the `packet_create()` function. It first checks whether the current process is capable to create raw packet sockets, for this is restricted to privileged users. Next it allocates a protocol-layer socket representation using the `sk_alloc()` function. It sets the protocol interface functions to use with the new BSD socket to the ones defined by the packet protocol family, initializes the allocated `struct sock` object and connects it to the BSD socket using the `sock_init_data()` function. Afterwards a protocol-private data structure is allocated and connected to the protocol-layer socket object and the socket's destructor is set to the one defined by the packet protocol family. If the user specified a protocol to the `socket()` call, the creation function installs a packet receiving hook for the current socket and the specified protocol. Finally, the

allocated socket object is inserted into a list of all packet sockets, which is used by the installed network device notifier and the `procs` operations to iterate over all packet sockets alive.

After the protocol family has initialized its socket object, `sys_socket()` calls the `sock_map_fd()` function that gets an unused file descriptor from the file descriptor list of the current process and an empty `struct file` object. After the file object has been initialized and connected to the socket and the file descriptor, the descriptor is returned to the user to be used as the socket's identifier in subsequent socket function calls.

2.3.3 Receiving a Data Frame

The reception of network data is accomplished in two steps: First, the network interface card and its driver take the asynchronously arriving data frames from the network and hand them to the networking subsystem where the target protocol enqueues the data to the receive queue of the determined target socket. Second, the user space application requests the received data using any of the receiving socket calls, as it is described in the next section. *Figure 2.3.2* shows an overview over the whole reception process.

Network data frames may arrive asynchronously and unexpectedly at the network interface card (NIC). Therefore the NIC and its device driver must be prepared to receive the data and store it into memory buffers.

The Intel E100, Intel E1000 and Broadcom BCM4401 network interface cards contained in the development and test systems use direct memory access (DMA) and a special data structure in the system's main memory, called DMA ring, to accomplish this task. The DMA ring consists of a circular array or list of DMA descriptors² and corresponding data memory buffers, as depicted in *Figure 2.3.1*. Upon interface initialization the device driver sets up the DMA descriptors and data buffers of the ring and writes its base address into the appropriate register of the network interface card. On receipt of a data frame the DMA controller of the NIC stores the data into the next empty buffer of the DMA ring and marks this buffer occupied.

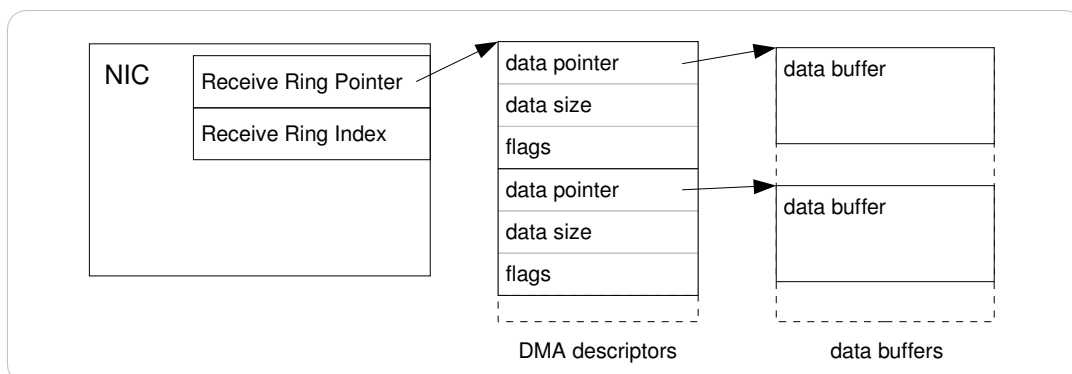


Figure 2.3.1: DMA ring (exemplary)

Basically, there exist two different techniques to process asynchronous events in computer systems. While *interrupts* yield low latencies for low rates of incoming data frames, the system can be used 100% to process interrupts under heavy network load, therefore starving any user-space processing. On the other hand, *polling* can be used to

² The exact structure of the DMA descriptors and the involved interface registers are NIC specific

query the NIC for arrived data in a tight loop, abusing the CPU for “busy waiting” when there's no data to receive, or in predefined intervals leading to an increased latency, especially for light loads.

The Linux networking subsystem has used the *New API* (NAPI) [2] for network device drivers since kernel version 2.4.20 to utilize the benefits and minimize the drawbacks of both the interrupt and polling techniques. NAPI incorporates a mixture of them in the following way: Only the reception of the first data frame triggers an interrupt causing the interrupt service routine of the device driver to be executed. Consecutively received data frames will then be handled using polling.

Although the interrupt service routine is driver-specific, it performs the following common actions:

- 1 calls `netif_rx_schedule_prep()` to check whether the device can be enqueued for polling, and if so
- 2 disables all hardware interrupts for the NIC associated with data reception
- 3 calls `__netif_rx_schedule()`, which in turn does the following:
 - 3.1 enqueues the network device into the polling list of the per-cpu data structure `softnet_data`
 - 3.2 invokes `__raise_softirq_irqoff()` to schedule the activation of the `NET_RX_SOFTIRQ` software interrupt

The hardware interrupt service routine of the device driver exits at this point, keeping the time spent with hardware interrupts disabled as short as possible. The more time expensive operations have been deferred to the next activation of the software interrupt handler for network data reception.

Software interrupts, or *softirqs*, are the Linux kernel's means to implement deferred functions. In version 2.6 Linux kernels six different softirqs are defined, two of which are dedicated to network data receiving and sending. The processing of softirqs is activated at different points of the kernel code, especially

- from the `irq_exit()` macro, which is called when the handling of a hardware interrupt is finished to ensure that the deferred part of the interrupt handling is activated as soon as possible
- when the special per-cpu *ksoftirqd* kernel thread is executed.

When a `NET_RX_SOFTIRQ` software interrupt is pending, the associated request handler `net_rx_action()` is called by the main softirq dispatching function `__do_softirq()`. It sequentially processes the devices queued for polling and executes the polling functions registered by their device drivers. The main tasks of a polling function are to form socket buffer structures from the received data, set their protocol fields accordingly for the next higher level network protocol and hand them to the networking subsystem. The target protocol is determined from the packet type field contained in the header of the received packet using an appropriate header parsing function, like `eth_type_trans()`. The delivery to the networking subsystem is performed using `netif_receive_skb()`. A per-device quota prevents the drivers from delivering arbitrary numbers of packets to the kernel thus starving other competing devices and an

overall budget limits the number of packets that can be delivered by a single activation of the network receive software interrupt handler. When this budget is exceeded the handler yields to other system threads after having scheduled a new invocation.

To form the socket buffers some different approaches are used. First, a new socket buffer of the required size could be allocated and the received data could be copied into it. Second, the data buffer of the receive ring could already be part of a socket buffer. In this case it would only have to be unlinked from the receive ring before being delivered to the kernel. The now free place must then eventually be refilled with a newly allocated empty buffer. This approach is used by the device drivers of the Intel E100 and E1000 network interface cards contained in the development and testing systems. The third approach combines the copying and replacing ones by using a copy threshold value: If only a few bytes have been received, the copying approach is used while the replacing approach is used whenever the amount of received data exceeds the copy threshold value. This procedure is used by the device driver of the Broadcom BCM4401 network interface card.

When there is no more data pending the driver's polling function calls `netif_rx_complete()` to remove the network device from the polling list. Afterwards it re-enables the NIC's receiving hardware interrupts to wait for the next data frame to arrive.

The function `netif_receive_skb()` represents the transition from the network device layer to the protocol stack. It is also the lowest point inside the networking subsystem that provides a hook to install custom code without the need to adapt a network device driver or the kernel code itself. A protocol interested in the reception of network data can register a packet handler callback function as described in 2.3.1 that will be executed whenever relevant data arrives.

The function `netif_receive_skb()` basically performs the following actions:

1. It sets the receiving timestamp of the socket buffer (if not set already) and adjusts the higher level protocol header pointers to the beginning of the packet data block.
2. It calls every packet handler registered for all packet types using `deliver_skb()`.
3. It calls `handle_diverter()` and `handle_bridge()` to perform any configured frame diverting or bridging.
4. If the socket buffer has not been consumed by the bridging code it calls the handlers registered for the actual type of the packet indicated by the socket buffer's protocol field using `deliver_skb()`.

By this procedure a single socket buffer may be passed to multiple packet handlers, e.g. a packet sniffer working in promiscuous mode in addition to either the bridging code or the target network protocol handler. To minimize the processing overhead the function `deliver_skb()` only passes a reference to the socket buffer containing the packet data to every packet handler after having incremented its reference count. Therefore the handler must be aware of the fact that the socket buffer might be shared. If it wants to change any value of the buffer's descriptor, e.g. adjust the header pointers or make use of the private data area, it has to create a private clone of it. This holds even true if the handler only wants to enqueue the buffer to some queue, since a single socket buffer can only be in one queue at a time. Otherwise the packet queue of another handler might be

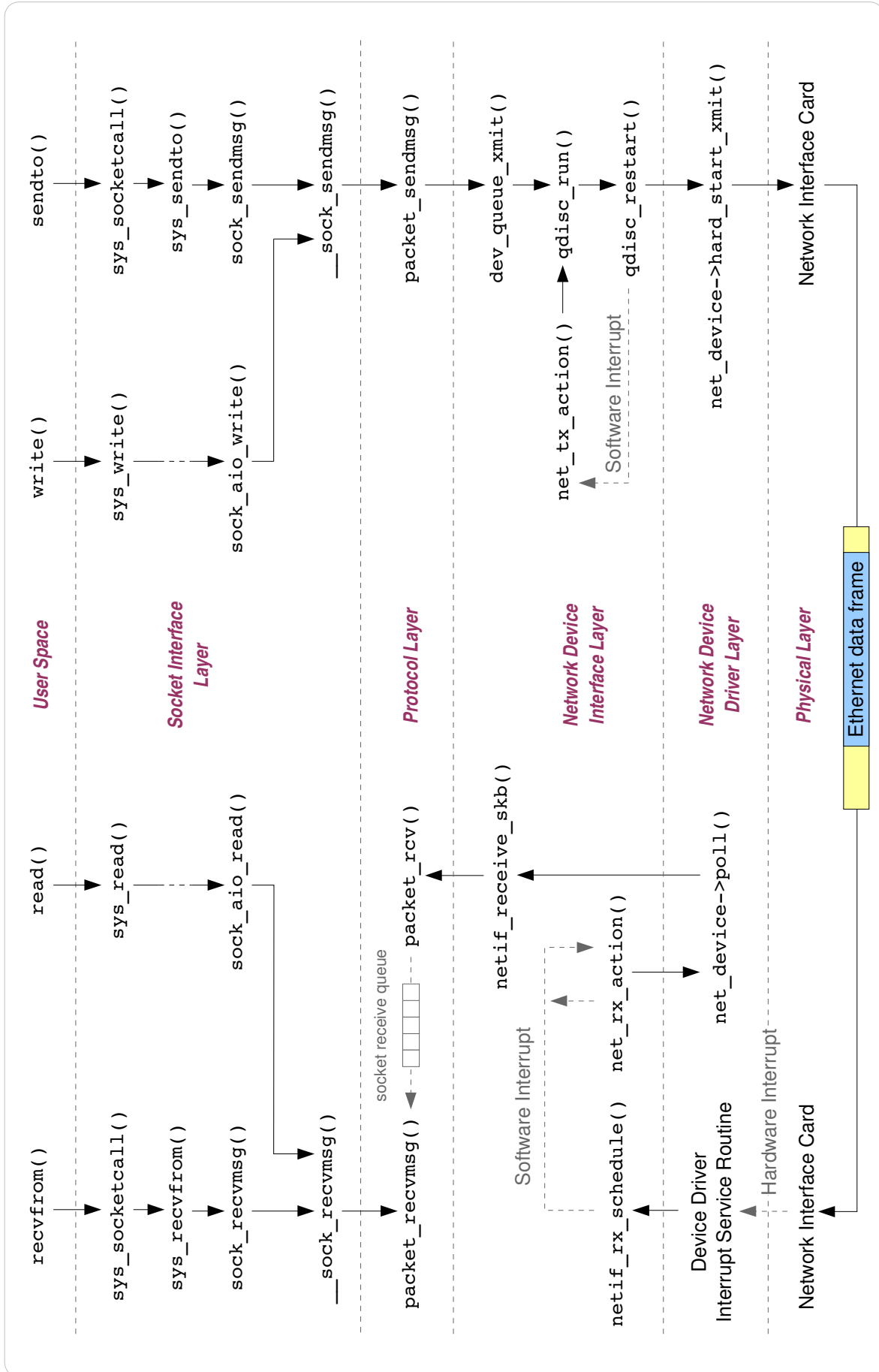


Figure 2.3.2: Data Transmission and Reception Through the Linux Networking Subsystem

torn apart by re-enqueuing the buffer. If a protocol handler wants to change the data contained in a buffer, it has to create a full private copy of it to prevent other handlers working with the same buffer from also witnessing the changes, which could lead to unpredictable results.

The packet handler function registered by datagram packet sockets is `packet_rcv()`. It receives the socket buffer reference, the receiving network device and the receive hook data structure used during the installation of the hook as parameters. Its first task is to determine the socket the packet should be delivered to. Since each raw packet socket installs its own receiving kernel hook upon creation or binding and puts a reference to itself into the hook data structure this turns out as simple as re-reading the value from the passed parameter. In contradiction other protocols, e.g. those from the internet protocol family, have to lookup the socket, e.g. from a hashtable by the source and destination addresses and ports.

Next the handler runs the binary packet filter associated with the socket, if any. The binary packet filter is an array of assembler-like instructions attached to the socket's `sk_filter` field which are interpreted by the function `sk_run_filter()` that emulates an accumulator machine. The instructions allow the data inside the socket buffer to be examined and simple calculations to be performed. The return value of the filter is the length the packet data should be trimmed to. Afterwards the packet socket's receive handler checks whether there is enough receive buffer space left to accept the packet. If not or if the packet filter returned zero the packet is discarded.

The check for a shared socket buffer has been deferred until now hoping that many of the packets are dropped by the filter or receive buffer check. While the benefit of this procedure is that only an acceptable shared socket buffer has to be cloned now, the drawback is that all changes to its descriptor performed until now must be undone before it is discarded.

Next the remote address information is stored into the socket buffers protocol-private data area. This is especially necessary for any information depending on the input device of the packet. Because later on when the user requests the received data and remote address, e.g. using `recvfrom()`, the device might already be gone. To determine the link layer address and its length from the received data packet the `dev->hard_header_parse()` function of the device descriptor is used, which points for example to the `eth_header_parse()` function for Ethernet devices.

Then the socket buffer is enqueued to the socket's receive queue after it has been trimmed to the size indicated earlier by the binary packet filter and handed to the target socket deducting its true size from the socket's receive buffer memory using the `skb_set_owner_r()` function.

Finally any processes waiting on the socket for data are notified using its event notifying function `sk->sk_data_ready()` before the protocol packet handler and the processing of the network data reception software interrupt ends.

2.3.4 Requesting a Received Packet

There are two slightly different ways for user space applications to request data received on a network socket. On the one hand the data reception socket calls `recv()`, `recvfrom()` or `recvmsg()` can be used. On the other hand it is also possible to invoke the standard file reading functions `read()` or `readv()`, since the socket is closely connected to a file of the `sockfs` special file system and represented by that file's descriptor in user mode.

All socket calls are translated into the `sys_socketcall()` system call by the standard C library which serves as a multiplexer and invokes the `sys_recv()`, `sys_recvfrom()` or `sys_recvmsg()` functions depending on the actual data reception socket call performed. All of these functions have the same layout: First, they call `sockfd_lookup()` to determine the socket associated to the file descriptor given by the user. Second, they copy some of the parameters provided by the user into kernel space, perform some necessary parameter checks and prepare the parameters needed for the final call of the `sock_recvmsg()` function, that in turn calls `__sock_recvmsg()`.

The file reading functions take their way through the virtual file system (VFS) first, which is described in detail in [3]³. The `read()` function, for example, invokes the `sys_read()` system call, that first determines the `struct file` object specified by the provided integer file descriptor. After some parameter checks and preparations have been performed, the `aio_read()` function of the file object's operations is called, that has been set to the `sock_aio_read()` function of the `sockfs` special file system upon the creation of the socket and its related file. This function in turn calls `__sock_recvmsg()`, where the two execution paths join.

The `__sock_recvmsg()` function calls the `recvmsg()` member of the BSD socket's interface functions. For raw packet sockets this field points to `packet_rcvmsg()`.

The basic task of a protocol's data reception interface function is to fetch pending socket buffers from the socket's receive queue and to copy their contents to a buffer specified by the user space application. Therefore, after having checked the provided parameters, `packet_rcvmsg()` invokes the generic datagram reception function `skb_recv_datagram()` that tries to fetch the first socket buffer contained in the socket's receive queue. The function will suspend the current process for the socket's receive timeout, if it is invoked in blocking mode and there are no buffers in the queue. If any data was received, it is copied to the provided user data buffer using the `skb_copy_datagram_iovec()` function. If the user supplied a buffer smaller than the received data packet, the excessive data is discarded and the `MSG_TRUNC` flag is set to inform the user about the data loss. Afterwards the source address of the received packet is copied from its temporary storage in the socket buffer descriptor's private data area to the buffer the `msg_name` field of the `struct msghdr` parameter points to. The kernel will copy the address to the user when the protocol-dependent data reception function ends successfully. Finally the socket buffer is released using `skb_free_datagram()`. On success the `packet_rcvmsg()` function returns the number of data bytes copied to the user.

³ The book actually covers version 2.4. Linux kernels, but can still be applied to version 2.6. kernels to a large extent.

2.3.5 Sending a Data Frame

Similar to the reception of network data the transmission can also be initiated in two slightly different ways. On the one hand the data transmission socket calls `send()`, `sendto()` or `sendmsg()` can be used. On the other hand it is also possible to invoke the standard file writing functions `write()` or `writew()` on the file of the `sockfs` special file system associated to the socket. *Figure 2.3.2* shows an overview over the whole transmission process.

The standard C library translates the socket calls into the `sys_socketcall()` system call which serves as a multiplexer and invokes the `sys_send()`, `sys_sendto()` or `sys_sendmsg()` functions depending on the actual data transmission socket call performed. All of these functions have the same layout: First, they call `sockfd_lookup()` to determine the socket associated to the file descriptor given by the user. Second, they copy some of the parameters provided by the user into kernel space, perform some necessary parameter checks and prepare the parameters needed for the final call of the `sock_sendmsg()` function, that in turn calls `__sock_sendmsg()`.

The file writing functions take their way through the virtual file system (VFS) first. The `write()` function, for example, invokes the `sys_write()` system call, that first determines the `struct file` object specified by the provided integer file descriptor. After some parameter checks and preparations have been performed, the `aio_write()` function of the file object's operations is called, that has been set to the `sock_aio_write()` function of the `sockfs` special file system upon the creation of the socket and its related file. This function in turn calls `__sock_sendmsg()`, where the two execution paths join.

The `__sock_sendmsg()` function calls the `sendmsg()` member of the BSD socket's interface functions. For raw packet sockets this field points to `packet_sendmsg()`.

The main tasks of a protocol's data transmission interface function are to form socket buffers from the data provided by the user space application and hand them to the device interface layer to be queued for transmission.

First, the `packet_sendmsg()` function determines the destination address to use for the packet about to be created. If no addressing information was provided to the function, e.g. the application called `send()`, `write()` or `writew()`, it tries to use the remote address stored in the socket's private data area during a `connect()` socket call. Next the descriptor of the sending device is determined from the interface index contained in the addressing information using `dev_get_by_index()`. It is needed at first to ensure that the size of the data to be sent does not exceed the maximum transfer unit (MTU) of the device.

To allocate a new socket buffer the `sock_alloc_send_skb()` function is invoked, which in turn calls `sock_alloc_send_skb()`, that performs the following actions:

1. It checks whether the socket has enough write buffer space left. If not and if the function was called in blocking mode, it will suspend the current process for the time specified by the socket's send timeout.
2. It allocates the socket buffer using the `alloc_skb()` function.

3. It reduces the socket's write buffer space by the socket buffer's memory using `skb_set_owner_w()`.

Before the hardware header can be filled into the socket buffer using the `dev->hard_header()` function of the descriptor of the outgoing device, an appropriate amount of space needs to be reserved inside the buffer's data area. For that purpose `skb_reserve()` is called using the `LL_RESERVED_SPACE()` macro to determine the amount of header space needed by the device.

Next the user data is copied into the socket buffer using the `memcpy_fromiovec()` function and a few remaining fields are set:

- the `protocol` field is set to the requested packet type
- the `dev` field is set to the descriptor of the outgoing device
- the `priority` field is set to value of the socket's field `sk_priority`

If the network device chosen for sending is up, the prepared socket buffer is finally handed to the device interface layer by a call to the `dev_queue_xmit()` function. It will return a negative error code on error, `NET_XMIT_SUCCESS (0)` on full success or a positive result code describing what happened to the socket buffer. The `net_xmit_errno()` macro is used to determine the error code to return to the user from the result code. But even if `dev_queue_xmit()` returned success this is no guarantee that the socket buffer will be transmitted as it can still be dropped due to local congestion or traffic shaping.

The `packet_sendmsg()` function will return the number of bytes just queued for transmission on success.

At the beginning the `dev_queue_xmit()` function checks whether the outgoing device and its driver support all of the special features the socket buffer bears: If a list of fragments is appended to a socket buffer, the device driver must be able to handle it, specified by the `NETIF_F_FRAGLIST` flag in the `features` field of the device descriptor. If additional pages of memory are appended to the socket buffer, the device needs to be able to perform scatter-gather i/o, which is specified by the `NETIF_F_SG` flag. Additionally, if any of the appended pages resides in high memory, the device must be able to reach high memory pages, specified by the `NETIF_F_HIGHDMA` flag.

If the outgoing device and its driver don't support all of the needed features, the socket buffer will be linearized using the `__skb_linearize()` function, i.e. a copy will be created which contains all the data in the socket buffer's consecutive data area.

Following, the `dev_queue_xmit()` function determines the queuing discipline of the network device and invokes its `enqueue()` function to enqueue the socket buffer. If the queuing discipline doesn't offer an `enqueue()` function, which is common for software devices like loopback or tunnels, the socket buffer is handed directly to the device driver, as described later on. Afterwards `qdisc_run()` is invoked that will keep on processing the queue using `qdisc_restart()` as long as it is not stopped, throttled or empty.

The `qdisc_restart()` function upon each invocation performs the following actions:

1. It dequeues a socket buffer from the queuing discipline using its `dequeue()` function.

2. It hands a clone of it to any data receiving hooks registered in the queue for all packets (cf. 2.3.1) using the `dev_queue_xmit_nit()` function.
3. It hands the socket buffer to the `dev->hard_start_xmit()` function of the descriptor of the outgoing device that points to the transmission function of its device driver.
4. If the driver's transmission function doesn't indicate successful transmission, it hands the socket buffer to the queuing discipline to be re-enqueued. Afterwards it calls `netif_schedule()` that enqueues the network device into the `output_queue` list of the per-cpu data structure `softnet_data` and invokes `__raise_softirq_irqoff()` to schedule the activation of the `NET_TX_SOFTIRQ` software interrupt.

Similar to the receive ring (cf. 2.3.3) the Intel E100, Intel E1000 and Broadcom BCM4401 network interface cards contained in the development and test systems use a circular array or list of DMA descriptors, the transmission ring, to manage the physical transmission of data. When the hardware transmission function of the driver is invoked, it enters the physical address and size of the data contained in the provided socket buffer into the next free transmission DMA descriptor. If a list of fragments or additional memory pages are appended to the socket buffer (and the driver supports those features) they are processed the same way. Afterwards the network interface controller is notified about the pending transmission requests by writing into its memory-mapped control registers. Whenever there are no more free transmission descriptors the driver will stop the transmission queue using the `netif_stop_queue()` function to prevent new socket buffers from being delivered for transmission.

After the physical transmission has completed the network interface card will generate a hardware interrupt to notify the device driver. The driver's interrupt service routine will take the processed socket buffers off the transmission ring and hand them to `dev_kfree_skb_irq()`. This function enqueues them to the `completion_queue` member of the per-cpu data structure `softnet_data`. If the transmission queue of the device was stopped the interrupt routine wakes it up using `netif_wake_queue()` and enqueues the network device into the `output_queue` list of the per-cpu data structure `softnet_data`. In both cases the activation of the `NET_TX_SOFTIRQ` software interrupt is scheduled.

If a `NET_TX_SOFTIRQ` software interrupt is pending, the associated request handler `net_tx_action()` will be called when software interrupts are processed the next time. It fulfils two tasks: First, it releases any socket buffer of the `completion_queue` list of the per-cpu data structure `softnet_data` by handing it to `__kfree_skb()`. Second, it invokes the `qdisc_run()` function for every device of its `output_queue` list to restart the processing of its transmission queue.

Chapter 3: Evaluation of Existing Solutions

3.1 U-Net: A User-Level Network Interface for Parallel and Distributed Computing

U-Net [4] completely removes the kernel from the communication path, while still providing full protection to ensure that applications can't interfere with each other. Therefore it provides user-space processes with a virtual view of network interfaces to enable user-level access to high-speed communication devices. This allows for the construction of protocols at user level whose performance is only limited by the capabilities of the network. The process in turn has full control over the contents of the messages but it is also responsible for the management of the send and receive resources, e.g. memory buffers.

The operation of U-Net is based upon three data structures: A process that wants to access the network first creates one or more endpoints, i.e. handles into the network. Afterwards it associates a memory region that can hold the message data, called communication segment, and queues for send, receive or free message descriptors with the endpoint. The communication segment is typically pinned down to physical memory to prevent it from being moved, e.g. to swap space. The send and receive queues hold descriptors with information about the source and destination endpoint addresses of messages, their size and offset within the communication segment. The free queue holds descriptors for free data buffers to be used by the network device for further receives.

To perform a sending operation, the application has to compose the data in the communication segment and queue a descriptor for the message to the send queue. The device is then expected to pick the message up and put it onto the network. Incoming messages are distributed by the U-Net multiplexing agent based on their destination. The received data is transferred into the determined communication segment and a descriptor is queued to the corresponding receive queue. The application can poll for incoming data by periodically checking the status of the receive queue, it can block waiting for the next message to arrive or it can register a callback function to get notified by U-Net.

U-Net does not provide any reliability beyond that of the underlying network. Thus, using Ethernet, messages might be lost. To increase the applicability of U-Net the internet protocol family containing the standard protocols TCP, UDP and IP has been implemented for it.

Since U-Net expects the network device to be able to handle the used message descriptors, it is limited to a small range of devices providing this feature, or at least requires adapted device drivers that are able to work with them.

The development of U-Net has been cancelled, as it has been superseded by the virtual interface architecture.

3.2 M-VIA: A Modular Implementation of the Virtual Interface Architecture

The Virtual Interface Architecture (VIA) [5] is an industry standard for high performance communication on system area networks. It provides applications with direct but protected access to network interface cards. This way the processing overhead of system calls or the TCP/IP protocol stack can be avoided while the applications are still prevented from interfering with each other. The implementation of VIA consists of three components: A user-level library, a kernel agent that assists during the setup phase of the protected connections and the network interface card itself.

To set up a communication the application needs to allocate a virtual interface (VI) that serves as a handle to the network. A send and receive queue is associated to each VI. To send data, message descriptors for data buffers are appended to the send queue. The VIA will set a completion flag in the descriptor after having transmitted the message. To receive data, descriptors for empty buffers are appended to the reception queue. Each VI is connected to a single remote VI. But applications can create completion queues to be filled with descriptors for completed data transfers and associate them with multiple virtual interfaces. All memory used for a communication, including queues, descriptors and buffers, must be registered with the VIA first. It manages a private user translation lookaside buffer (UTLB) to assist with virtual address translations and pins down the registered memory to prevent it from being swapped out.

Modular-VIA (M-VIA) [6] is a modular implementation of the VIA standard for Linux written at the NERSC center at Lawrence Berkeley National Laboratory. Besides a user-level library and a loadable kernel agent module it contains several modified device drivers. The latter are needed for network interface cards that don't natively support the VIA standard. Therefore its use is limited to network devices with native VIA support or devices covered by the provided drivers.

Data is sent using a fast system trap to privileged code and received by low level interrupt handlers inside the kernel (the technique used by U-Net). This way zero-copy sends and single-copy receives are achieved. Post version 1.2b2 M-VIA releases provide reliable communication even upon natively unreliable networks, like Ethernet, by means of acknowledgements and retransmissions.

To provide a common programming interface for parallel applications the MPI library MPICH [7] has been ported atop M-VIA by the MVICH [8] project.

The development of M-VIA has been cancelled in 2002, when the funding of the project ended. The last releases were M-VIA version 1.2 (9/30/02) and MVICH 1.0 (8/21/01).

3.3 Bobnet: High Performance Message Passing for Commodity Networking Components

Bobnet [9] provides zero-copy sending directly from the user data buffers and single-copy receiving. It works on commodity networking components, e.g. generic Ethernet devices, as well as on faster devices, like Myrinet [10]. It provides a selectable reliability from none to full reception acknowledgement. Multiple higher level protocols, like TCP or the VIA standard are supported.

Bobnet is implemented using a user agent and a kernel level agent. It offers its services to user-level applications through an implementation of the MPLite [11] message passing library, that provides a crucial subset of all MPI calls satisfying most parallel applications. The library translates all calls into calls of the user agent, the functions of which are designed to match the virtual interface architecture (VIA) [5] standard. To send or receive data the user agent creates and enqueues descriptors carrying information about the sizes and memory locations of buffers used as sources or targets of the data. Afterwards it makes a fast trap into the kernel to inform the kernel agent. This agent again translates the sending descriptors to the native data structures needed by the device driver, queues them to the driver and initiates the hardware sending process. When Bobnet data is received by the driver, it will invoke the kernel agent that searches the queue of receive descriptors for a matching pre-posted receive and copies the data to the specified user buffer.

The development of Bobnet has been cancelled in 2000, as the authors have moved on to other projects.

3.4 GAMMA: The Genoa Active Message MACHine

GAMMA [12, 13] is based on the Active Message communication paradigm [14]. Active Messages eliminate the need of data copies along the communication path. This is achieved by letting each arriving message trigger the execution of a receive handler function of the destination process that immediately consumes it, i.e. integrates it into the current computation.

GAMMA is implemented as a custom NIC device driver and offers its services to the user level through additional system calls. Its communication services are made available to user applications through a programming interface implemented as a user-level library that exploits the additional system calls. Both C and FORTRAN programming interfaces have been developed.

Each parallel application can use up to 254 ports that can be

- connected to the parallel PID, node and port of a remote destination process for any data sent through the port
- bound to a local data buffer and receive handler function for any incoming data.

This way the destination is completely defined by specifying the local port number when sending data. When receiving data, the activity of storing incoming messages into their destination buffers can directly be performed by the GAMMA device driver rather than by

a user-defined receiving function, because the kernel knows about the local binding of any port. Afterwards the receive handler is called, that can bind the port to a new buffer for subsequent receives. Besides uni-cast transmissions GAMMA also supports group-based broadcasts, that are efficiently handled using the native Ethernet hardware broadcast service.

GAMMA is highly optimized: It uses light-weight system calls, that save only a subset of the machine registers and do not invoke the scheduler upon return, and an optimized code path to the interrupt handler of the network device driver. The receiver can poll actively for incoming messages in order to anticipate their reception as much as possible. When sending data, GAMMA uses pre-computed Ethernet headers built when the port was bound to a destination.

The implementation of the GAMMA device driver introduces a restriction: It can manage IP traffic only, as long as no parallel job is running. This prevents any auxiliary communication over the interface used by GAMMA during the execution time of the job.

To provide a common programming interface for parallel applications the MPI library MPICH [7] has been ported atop GAMMA by the MPI/GAMMA [15, 16] project.

As GAMMA is implemented as a device driver, it only supports a small range of the available network interfaces.

3.5 EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing

EMP [17] uses the special features of NICs based upon the Alteon Tigon2 chipset to implement a reliable low-latency high-performance communication means. The chipset includes an ASIC with two MIPS-like microprocessors running at 88 MHz and external SRAM memory. Therefore, all parts of the messaging system, like descriptor-management, data fragmentation and reliability, could be implemented on the programmable NIC.

Sending and receiving operations are initiated, checked and finalized by passing descriptors between the host system and the NIC. EMP avoids all buffering overhead. During sending the network device copies MTU-sized chunks from the user-space buffer using DMA. Unexpectedly received data frames, i.e. not matching any pre-posted receive request, are immediately discarded. EMP supports flow control, acknowledgements and retransmissions making the communication reliable.

As EMP relies on the special features of the Alteon Tigon2 chipset, it can only be used with network devices based on this chipset.

3.6 Evaluation Summary

Several approaches have been followed to reduce the message latency, especially focussed on system-area networks and cluster communication. Most of them aim at the minimum latency values possible. To achieve this goal the copying of data is avoided and the Linux kernel is removed partly or completely from the communication path.

This technique, however, introduces two side-effects. First, the network communication management, especially the allocation and registration of communication data buffers and the management of communication descriptors, is moved to the application to a large extent. Furthermore the communication services are mostly provided to the application by a special interface. Although these extra efforts can be hidden in a user-level library they may prevent a high performance protocol from being easily integrated into existing applications. Laying the memory management into the application's hand imposes another threat: When multiple processes want to access the network and register large communication buffers, the operating system might become unable to satisfy memory requests by swapping.

Second, the circumvention of the kernel and the use of special descriptor data structures raise the need for adapted network device drivers or devices capable of working with those structures. In this case the use of the high performance protocol is limited to the devices covered by the drivers provided with it. The use of device-specific features limits the range of usable network devices even more.

Chapter 4: Implementation of an Ethernet Datagram Protocol

The Linux kernel networking subsystem implements a highly efficient TCP/IP protocol stack. This and the reliability of the whole operating system make it well suitable for various kinds of networking applications, like routers or servers. But the standard TCP/IP protocol suite is not necessarily the best choice for system-area networking. It handles reliable communication but it also includes some features unneeded in cluster environments, like the additional addressing scheme and routing functionality.

As part of the communication latency is due to protocol processing it should be possible to reduce the message latency by just removing the TCP/IP protocol suite from the communication path. This way it would also be possible to maintain both, the standard socket interface for user-space applications as well as the kernel's default interface to the standard NIC device drivers.

By eliminating the TCP and IP protocols the following features are lost:

- *routing* of data packets between multiple physical subnets, including a separate end-to-end addressing scheme
- *fragmentation* of large messages to chunks fitting the MTU of the underlying medium
- port *multiplexing* allowing for several distinct communications to be lead concurrently via a single network interface
- *sequencing*, which ensures that all messages or parts of messages are received in the correct order
- *reliability*, which ensures that all data is delivered or an error is raised whenever this is not possible

To reduce the message latency to a minimum, only the features really necessary for cluster communication shall be implemented. Routing is unneeded in Ethernet cluster environments closely interconnected by a switched network while fragmentation might easily be performed in user-space. Sequencing and reliability shall be neglected in the first approach. Re-ordering should not occur in a switched Ethernet network providing only a single path between two network interface devices. And the bit error rate is stated in [16] to be less than 10^{-11} , leading to a frame loss rate of 10^{-7} , which in turn allows for error-free data transfers larger than 12 GB. Additionally performed transmission tests with an early version of the `netgauge` utility [18] using raw Ethernet packets didn't show any packet loss, either. The tool, however, provided only the possibility to perform tests with packet sizes up to the MTU of the underlying Ethernet protocol, i.e. 1500 bytes. Therefore `netgauge` will have to be extended in parallel to the development to a platform providing all the tests needed for the new protocol as well as for standard protocols, especially TCP, for comparisons.

The multiplexing feature, however, is necessary and cannot be omitted. The existing Open MPI [19] TCP communication module for instance uses a separate socket for each connection to a remote process. For a single parallel process per host this behaviour could be emulated in user-space by using the `sendto()` and `recvfrom()` functions and

examining the source address of each packet received. But especially for multi-processor architectures it could be desirable to execute multiple parallel processes on a single host that all want to access the network simultaneously and independent from each other. For that purpose an additional multiplexing facility is needed. Following the example of raw packet sockets, the packet type field of the Ethernet header could be used to separate the communications. But this approach has two drawbacks: First, multiple processes would have to agree on the distribution of the identifiers among each other, possibly requiring a central management. Second, the range of packet type identifiers is already partly used to specify which protocol data unit an Ethernet frame contains, e.g. an IP datagram. Therefore unrestricted use of this field could lead to incompatibility with other protocols.

This leads to the development of a datagram protocol atop Ethernet, providing a connection-less datagram transport with the necessary multiplexing facility implemented using ports.

4.1 The EDP Datagram Structure

The Ethernet Datagram Protocol (EDP), as shown in *Figure 4.1.1*, relies on the addressing and checksumming of the underlying Ethernet protocol and adds, similarly to UDP [20], fields for the source and destination ports and the length of the data payload. The port fields allow for communication multiplexing at the source and destination hosts, respectively. The data length field is mainly needed, because Ethernet pads out small frames to a minimum length to permit the execution of the CSMA/CD algorithm [21].

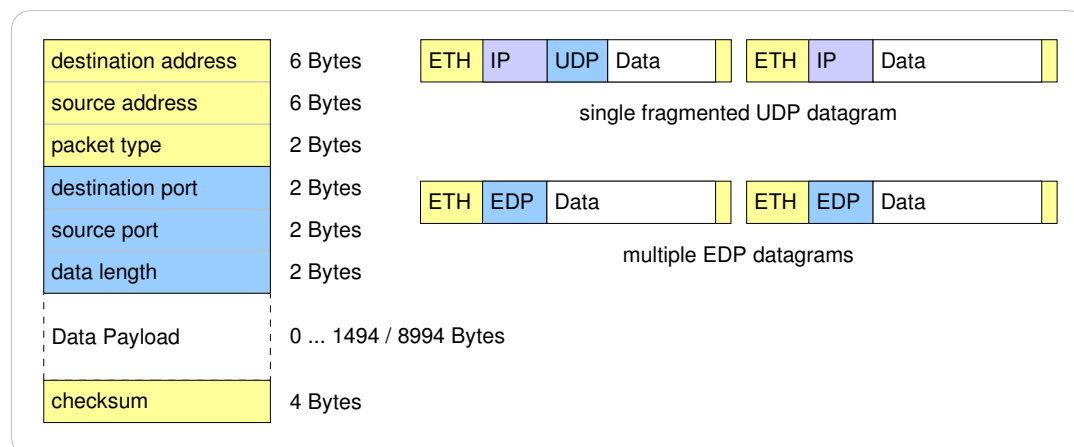


Figure 4.1.1: The EDP Datagram Structure

EDP introduces a very small protocol header consisting of only 6 bytes, which is less than a third of the minimum IP [22] header length of 20 bytes. Thus, assuming an MTU of 1500 bytes and not considering Ethernet protocol overhead, EDP would transmit 7 bytes for a one-byte message and 65,800 bytes for a 64 kB message. Compared to UDP, that would transmit 29 and 66,444 bytes in those cases, this is a reduction of about 76%, respectively 1%.

4.2 Implementation of the Ethernet Datagram Protocol

The implementation of the EDP protocol is mainly based on the implementation of raw packet sockets, described in detail in section 2.3, extended by port management and multiplexing inspired by the implementation of the UDP protocol.

4.2.1 Registering the Ethernet Datagram Protocol

The kernel module implementing the new Ethernet protocol family including EDP registers a single handler together with the `PF_ENET` numerical protocol family identifier for the creation of new sockets. Whenever the user requests the creation of a socket for the Ethernet protocol family the `sock_enet_create()` function will be invoked. It examines the requested socket type to decide which particular creation function has to be used. For datagram sockets, requested using the `SOCK_DGRAM` constant, the function `sock_edp_create()` will be called to perform the actual creation. This technique provides the possibility to add further protocols for different socket types in the future.

To be notified of relevant incoming packets EDP registers an own receiving hook for a single packet type identified by the `ETH_P_EDP` constant. Its value can easily be adjusted in the associated header file e.g. whenever interference with other protocols is noticed. Therefore basically any 16-bit number currently unassigned to any other protocol [23] can be used with an exception of the reserved range `0x0000 – 0x05ff`. Whenever the kernel receives an Ethernet frame containing a value out of this range in its packet type field, the `eth_type_trans()` function used to determine the packet type interprets it as the length specifier of an IEEE802.3 Ethernet frame [24], the structure of which is slightly different from the usual Ethernet V2.0 [25] frames.

4.2.2 The EDP Address Structure

The new Ethernet protocol family uses an own address structure of type `struct sockaddr_en` to communicate addressing information with user-space processes. It is a combination of the link layer socket address structure used by raw packet sockets and the socket address structure used by the internet protocol family. Besides the protocol family identifier, that should be set to the value of the `PF_ENET` constant, it contains fields for a port, the index of the outgoing interface, the length of the provided destination address and the destination address, itself. Depending on the use of the address structure the local or remote port has to be specified while other fields may have no meaning, like the destination address upon binding a socket. Currently the address structure is only used for Ethernet, so the address length should always contain a value of `ETH_ALEN` and the destination address should be set to an Ethernet address. But whenever EDP shall be ported atop another link-layer protocol, the length field can be used together with length-dynamic assignment and comparison functions to support differently sized addresses.

4.2.3 Binding a Local EDP Port

Before data can be sent or received, each socket needs to be bound to a local port and/or interface. When data is sent, the local port number is entered into the source port field of the EDP packet header to enable the recipient to answer the message. To receive data, e.g. the answer message, the socket must be bound to the destination port of the

message. Additionally, it could be desirable to receive only data from a particular network interface.

A socket can be bound manually to the desired interface and port using the `bind()` function. It takes a pointer to a socket address structure, in the case of EDP to the new Ethernet socket address structure filled with the index of the desired interface and the desired port. If zero is given for the interface index, the socket is bound to receive data from all available interfaces. If zero is given for the port, the protocol automatically chooses the next free one. Whenever a sending operation is performed on an unbound socket, it will be bound automatically to the outgoing interface indicated by the destination address and the next free port.

The bind handler of the Ethernet datagram protocol is the `sock_edp_bind()` function. At the beginning it performs some basic checks of the address structure provided by the user. If it contains a non-zero interface index, the descriptor of the specified device is determined next and the function checks whether the device is up. Afterwards the socket is locked to avoid simultaneous binds and the handler invokes the `edp_get_port()` function to assign the requested port to the socket.

EDP, like UDP, maintains an internal hashtable of bound sockets serving two basic purposes: First, it is used to determine whether a given port is already in use. Second, it is needed to efficiently determine the target socket for incoming data. The hashtable is protected by a read/write lock to prevent multiple sockets from acquiring the same port.

If the socket shall be bound automatically, the `edp_get_port()` function needs to determine the next suitable free port – suitable in a way, that the hashtable is utilized best. Therefore the function first searches the next empty hash bucket or, if no empty bucket exists, the one with the shortest list of already bound sockets. A port rover variable, that always indicates the last assigned port, is used to find a good starting point for the search. Next the function determines the smallest unused port number for the found bucket. Because it is possible to bind sockets directly to interfaces, a port is occupied, if:

- another socket is already bound to the requested port *AND*
- this other socket is bound to all interfaces *OR*
- the current socket tries to bind to all interfaces *OR*
- the current socket tries to bind to the same interface the other socket has already bound to

If the user requested the socket to be bound to a specific port, only the conditions above have to be checked. When a free port has been determined, the socket is inserted into the associated bucket of the hashtable and the local port and interface index are stored in the protocol-private socket data structure. They are used during sending operations to fill in the source port field of the packet header or to determine the outgoing interface and during receiving operations to find the socket matching the destination port and input interface of a datagram.

4.2.4 Sending an EDP Datagram

The EDP handler for sending requests is the `sock_edp_sendmsg()` function. It is derived from the send handler of raw packet sockets, described in detail in section 2.3.5, extended by the means needed for automatic binding and protocol header creation.

It first determines the destination address to use for the datagram: If an address structure has been provided to the function, i.e. the application used the `sendto()` call, the information contained therein is used. Otherwise, the function tries to fall back on the destination address stored in the protocol-private socket data structure during a previous `connect()` call. Next, the descriptor of the requested output device is determined and the function checks whether the device is up. If the socket has not been bound to a local port, yet, the send handler function invokes `edp_get_port()` to automatically assign a free port.

Since EDP is a datagram protocol, the `sock_edp_sendmsg()` function must ensure that the maximum possible message size is not exceeded. It can easily be determined from the MTU of the underlying Ethernet protocol and device reduced by the EDP packet header size. After having allocated a socket buffer of the appropriate size using `sock_alloc_send_skb()`, the send handler reserves the needed space in it for the Ethernet header using `skb_reserve()` and fills it in using the `dev->hard_header()` function of the device descriptor. Afterwards the EDP header is constructed from the destination and bound source ports and the data length. Then the data payload is copied into the socket buffer using the `memcpy_fromiovec()` function.

Finally the socket buffer is handed to the network device interface layer with a call to the `dev_queue_xmit()` function and the send handler returns the amount of bytes just transmitted.

4.2.5 Receiving Incoming EDP Datagrams

The EDP data reception hook is implemented by the `edp_rcv_hook()` function. It is derived from the reception hook of raw packet sockets, described in detail in section 2.3.3, extended by the means needed for port multiplexing.

At the beginning the receive hook ensures that the received packet is acceptable for the local host. It then invokes the `skb_share_check()` function to create a private clone of the socket buffer's descriptor, if it is shared with another handler. Afterwards `pskb_may_pull()` is called to check whether the socket buffer contains at least a complete EDP packet header. This function also handles fragmented or paged socket buffers correctly where part of the data may reside in external buffers or memory pages. If the socket buffer contains enough data, its head pointer is adjusted to the beginning of the data payload using `__skb_pull()`. Otherwise, the packet is discarded.

In the next step the EDP packet header is examined. Short EDP packets padded by Ethernet to its minimum frame length are trimmed to the size indicated by the header's length field. Its destination port field is used to determine the target socket for the packet. As no socket can be bound to port zero, this special value is used for control messages processed directly by EDP. Any other value is used to lookup the target socket from the hashtable of bound sockets.

If the found socket has enough receive buffer space left, the socket buffer is enqueued to its receive queue. But beforehand the source addressing information of the packet is stored into the private data area of the socket buffer's descriptor, especially those parts for which the descriptor of the input device is needed. This way it can later be re-read and delivered to the user when he requests the packet, e.g. with a call to `recvfrom()`, even if the input network device has already been shut down.

4.2.6 Additional Protocol Interface Functions Implemented by EDP

The `sock_edp_connect()` function implements the protocol's `connect()` handler. It just stores the provided remote address into the protocol-private socket data structure for use with subsequent data transmission requests, as it is usual for datagram protocols. If the socket has not been bound, yet, the function invokes `edp_get_port()` to automatically assign a free port. If on the other hand the socket has already been bound to a local interface, it must be prevented from being connected via another interface to keep the local port management consistent.

Together with the implementation of the `connect()` handler also the `getname()` handler, implemented by the `sock_edp_getname()` function, has been extended to return the remote address when invoked during a `getpeername()` call.

Two 'options' of EDP sockets can be obtained by a call to `getsockopt()` with a socket level of `SOL_EP`. The handler function `sock_edp_getsockopt()` will return a structure of type `struct edp_stats` containing the socket statistics about sent, received or dropped packets, if the `EDP_SO_STATS` option name is given. Additionally, a user-space application can query the protocol MTU of an EDP socket bound to a local device, i.e. the device MTU reduced by the size of an EDP packet header, using the `EDP_SO_MTU` option name.

4.3 EDP Micro-Benchmarks

The following *Figures 4.3.1 - 4.3.4* show the message latency and throughput values of the new EDP protocol as a function of the message size. They have been measured with the `netgauge` network benchmarking tool on test systems of both clusters by performing 2.500 test runs for each data size and calculating the minimum or maximum values.

On the test systems of cluster 1 the latency values could be reduced in comparison to TCP from 43 μ s to 30 μ s, i.e. by about 30%. Thereby the EDP protocol reaches latency values comparable to those of raw packet sockets used upon Ethernet. The influence of using non-blocking sockets on EDP is hardly noticeable in the latency diagram. Interestingly for TCP non-blocking sockets yield a latency increase of about 5 μ s. For very small message sizes up to 46 bytes constant or even declining latency values can be noticed, marking the area where the Ethernet hardware pads out the frames to a minimum size.

TCP reaches its maximum throughput of 600 Mbit/s on the test systems of cluster 1 at a data size of 65 kB, which coincides with the adjusted socket buffer size at that systems. For larger messages the throughput declines to about 500 Mbit/s as a result of the beginning retransmissions and the moderating of the implemented flow control algorithm. EDP exceeds both values and sustains at 720 Mbit/s over a large area. The effect of

using non-blocking datagram sockets becomes clearly visible. The omitted overhead of the scheduling operations performed on blocking sockets allows for data rates of up to 900 Mbit/s.

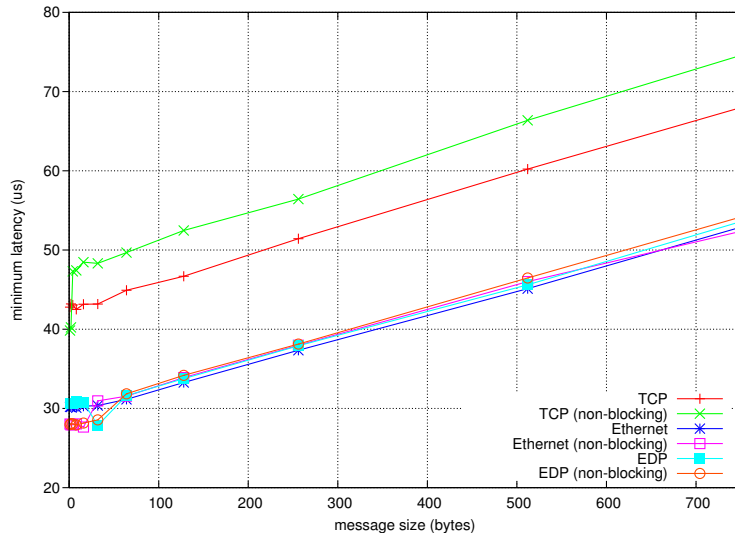


Figure 4.3.1: EDP Message Latency on Cluster-1

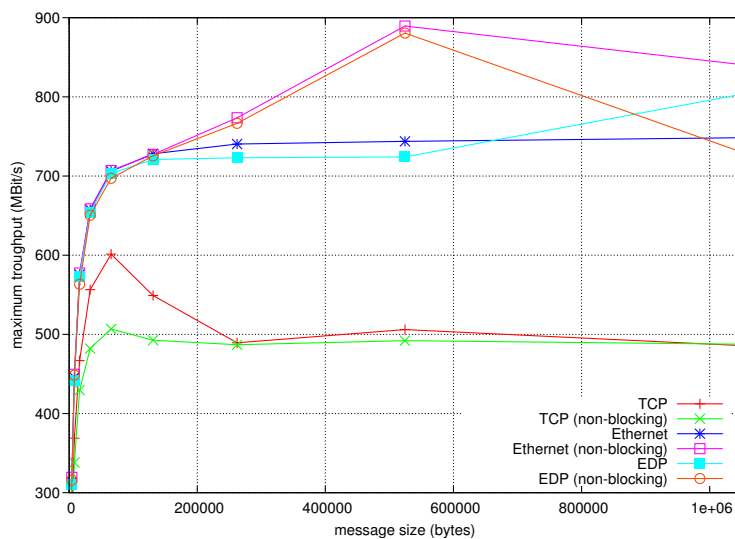


Figure 4.3.2: EDP Message Throughput on Cluster-1

The latency values measured for TCP on the test systems of the second cluster show an anomaly for very small message sizes between 1 and 8 bytes, a fluctuation between 40 μ s and 46 μ s. As all systems of both clusters use the same Linux kernel version the anomaly must be due to the different network devices or their drivers. EDP again reaches minimum latency values of 30 μ s. Switching the sockets to non-blocking mode the values could even be reduced by another 3 μ s by avoiding the process scheduling overhead. Thereby EDP again reaches the lower bounds set by raw packet sockets used upon the

Ethernet protocol. Considering the lower of the TCP latency values this yields a reduction of 25%, respectively 32.5%.

The higher processing power of the cluster 2 test systems also allows for a higher utilization of the bandwidth provided by the underlying Gigabit Ethernet. Therefore EDP reaches a maximum of about 850 Mbit/s or, with non-blocking sockets, 950 Mbit/s and stays only shortly behind the raw packet sockets. TCP in comparison, except for the anomalous peak, stays below 800 Mbit/s.

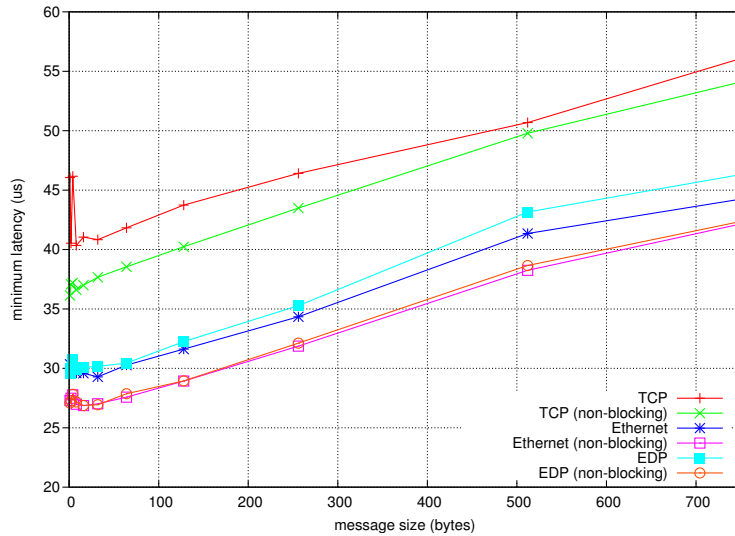


Figure 4.3.3: EDP Message Latency on Cluster-2

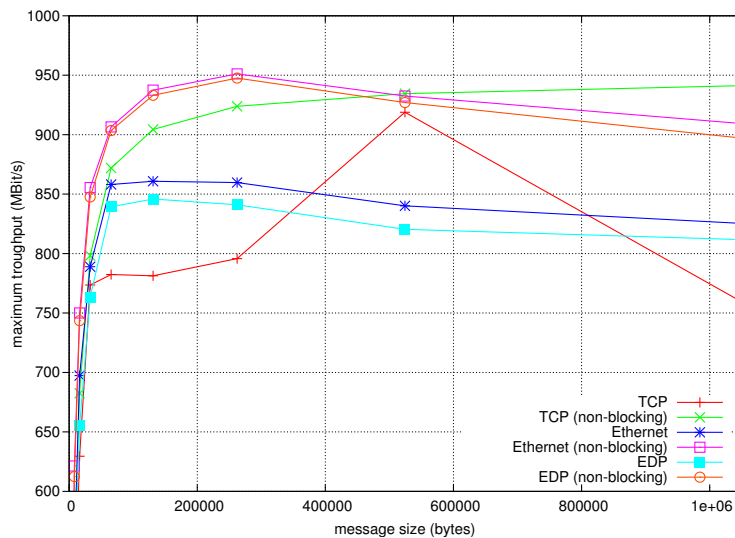


Figure 4.3.4: EDP Message Throughput on Cluster-2

4.4 Zero-Copy Data Transmission

Most of the existing solutions evaluated in section 3 try to avoid any data copying along the communication path to minimize the message latency. The prerequisite of this work to

use the standard Ethernet device drivers, however, implies two strong restrictions: First, the devices and their drivers will store every data packet received into the hosts main memory using DMA and hand them to the networking subsystem as socket buffers as described in detail in section 2.3.3. Therefore a true zero-copy reception is impossible to realize. The kernel itself avoids any data copying as it passes the socket buffer through the protocol layers, but at last one additional copying operation is necessary to store the data into the reception buffers of the user-space application. Second, the driver layer and the drivers themselves will only accept socket buffers for data transmission. Therefore any zero-copy sending approach is restricted to the means provided by socket buffers.

4.4.1 A Simple Zero-Copy Sending Approach

The basic idea of the approach followed here is to exploit the possibility to link additional memory pages to a socket buffer. This way the existing EDP sending interface function can be reused with a single modification: The `memcpy_fromiovec()` function used to copy the data from the user-space application buffers into the transmission socket buffer needs to be replaced by code that determines the needed page descriptors from the provided virtual addresses.

The memory management of the x86 processor family and the Linux operating system is described in detail in [3]³. In short, the Linux kernel uses descriptors of type `struct page` to manage the page frames a system's main memory is divided into.

So instead of copying the user data into the socket buffer the new EDP sending function needs to iterate over all elements of the provided i/o vector array, determine the page descriptor and offset from the virtual address of each element and enter those values into the socket buffer. Thereby it must be taken into account, that a single memory buffer could also cross the memory page boundaries, in which case an additional page would have to be appended to the socket buffer.

The standard kernel macros and functions for address translations, like `virt_to_page()` or `virt_to_phys()`, do only work correctly on the linear addresses used internally by the kernel. Therefore they cannot be applied to the virtual addresses supplied for data buffers of user-space processes. To determine the page frame descriptor from those addresses, the page tables of the process have to be traversed.

Pre-2.6.12 versions of the Linux kernel uses a three-level paging model with page global directory, page middle directory and a page table. In more recent kernel versions the paging model has been extended to four levels by a page upper directory, inserted between the global and middle directories. The kernel provides the macros `pgd_offset()`, `pud_offset()`, `pmd_offset()` and `pte_offset_map()` to determine the relevant entries from the single tables by a given virtual address. The page global directory of the current process, needed as starting point, can be determined from the memory management structure of the current process `current->mm`.

When the page frame descriptor has been determined, it is marked as being accessed using `mark_page_accessed()` and its reference count is incremented with a call to `get_page()` to indicate that it has a new user. Upon the destruction of the socket buffer, after it has been sent, the reference count of the page frame will be decremented again.

The `offset_in_page()` macro is now used to determine the starting point of the data inside the page from the given virtual address. Then the amount of the data residing inside the page is determined. Finally `skb_fill_page_desc()` is invoked to fill all the information into the next empty page slot of the socket buffer.

After all elements of the i/o vector array have been processed the new sending function continues similar to the old one by handing the socket buffer to `dev_queue_xmit()` to be queued for transmission. This kernel function will linearize the socket buffer, i.e. copy its fragments to a newly allocated consecutive buffer, if the target device or its driver do not support scatter-gather i/o or if any of the pages resides in a memory region not accessible by the NIC (cf. section 2.3.5).

4.4.2 Zero-Copy Sending Benchmarks

The following *Figures 4.1.1 - 4.4.4* show the message latency and throughput values measured for the zero-copy sending approach using the `netgauge` utility compared to the Raw Ethernet and TCP protocols. It is clearly to be seen that the zero-copy variant does not yield a noticeable latency improvement. In fact, it even increases the latency for small messages. This might be due to the fact, that now the NIC needs to perform at least two DMA operations for each data packet: one to get the Ethernet and EDP protocol headers from the socket buffer's data area and another one to get the payload data residing in an appended memory page representing a part of the user-space data buffer. Another issue might be that the user buffer is not necessarily aligned to a memory position optimal for the DMA.

For large messages the zero-copy approach causes a throughput reduction by about 3 % on the systems of cluster 1..

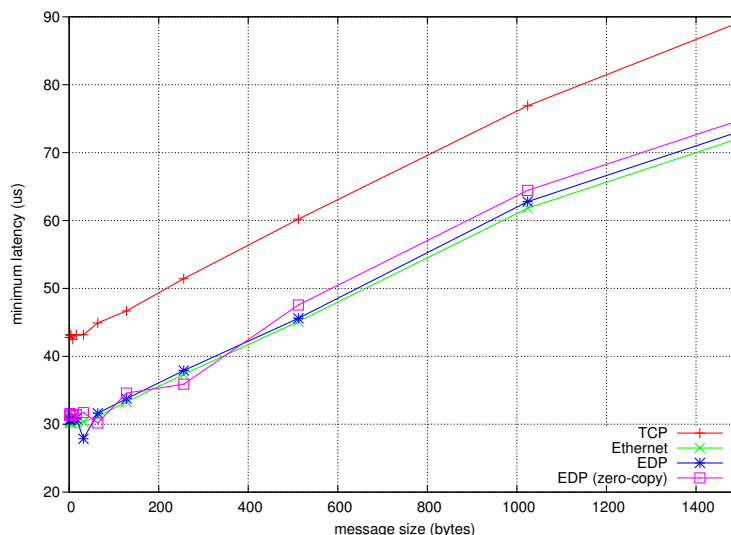


Figure 4.4.1: EDP Zero-copy Message Latency on Cluster-1

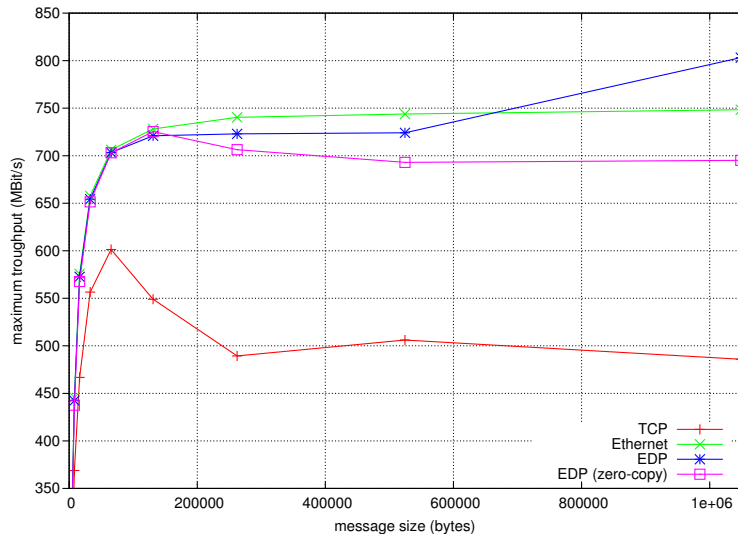


Figure 4.4.2: EDP Zero-copy Message Throughput on Cluster-1

The faster test systems of the second cluster showed the same behaviour, i.e. the zero-copy sending approach yielded higher latency values for small messages. The throughput of large messages on the other hand could be increased from about 820 Mbit/s to 831 Mbit/s, i.e. by about 1.3 %.

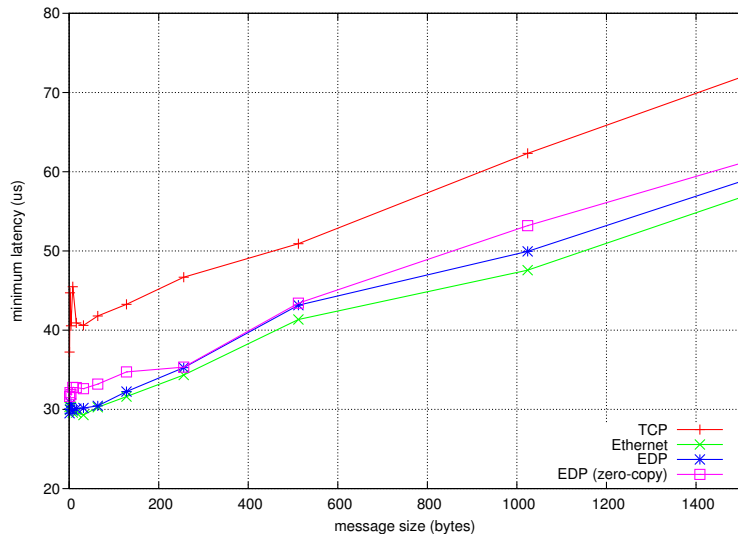


Figure 4.4.3: EDP Zero-copy Message Latency on Cluster-2

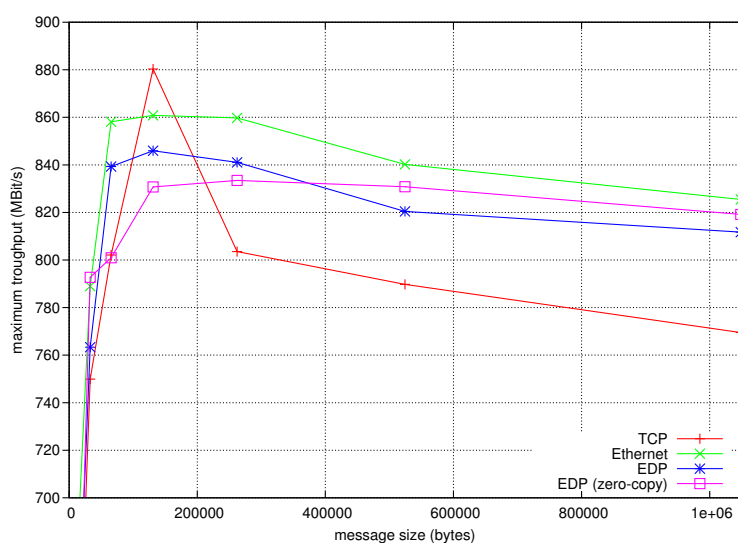


Figure 4.4.4: EDP Zero-copy Message Throughput on Cluster-2

4.4.3 Problems of the Zero-Copy Sending Approach

Between the moment the socket buffer is queued for sending and the completion of the transmission operation a certain amount of time may pass. The current simple approach ignores any changes that could happen to the user data buffer during that time. First, the involved pages are not write protected, so the application could overwrite the transmission buffer. Second, the involved pages are not locked in memory, so the operating system could choose to swap them out and re-use the gained space for other purposes. They are, however, marked as being accessed to inform the kernel that they are in use and *should* not be swapped out in the near future.

The problem here is not to set those page protection flags, as this could be done during the assembling of the socket buffer. It is a much more difficult task to determine when and to which value they should be restored. Basically, it is possible to attach a custom destructor function to the socket buffers that would be invoked upon its destruction, i.e. after the transmission has completed. But, due to the asynchronous nature of the transmission, it still could not tell, whether any of the pages has been involved in some other i/o processing or even another sending operation in the meantime that might have changed its flags.

Since the simple zero-copy approach did not yield a noticeable improvement of the communication latency or throughput it shall not be followed any further.

4.5 Conclusion

The initial considerations showed that it is not possible to use raw Ethernet as the only underlying protocol for parallel applications because of its lack of a communication multiplexing mechanism. Therefore a new light-weight datagram protocol has been developed atop Ethernet, incorporating the slim transmission and reception functions of raw packet sockets as well as a communication multiplexing inspired by UDP. The

benchmarks using the extended `netgauge` utility showed that a latency improvement for small messages of about 30% is possible by this approach. Additionally an enhancement of the throughput by 6% to 20% could be noticed. An additional zero-copy transmission approach introduced new problems but did not further improve the latency or throughput values.

But a datagram protocol does only allow for an unreliable transport of data packets. Therefore packet losses could be observed during the benchmarks, especially when higher message sizes required several packets to be sent in rapid succession. A packet can be lost due to one of the following reasons:

1. A host system with high processing power could generate more data packets over a certain time period than the local network interface is able to transmit. The device and its driver maintain a transmission ring buffer and the associated queuing discipline holds an additional queue to handle short bursts of packets. But if the high packet rate sustains over a longer period of time, all of the buffers will run full. The `dev_queue_xmit()` function will return an error code if a socket buffer could not be enqueued for transmission due to local congestion.
2. If the queuing discipline associated to a network device performs traffic shaping or policing, a packet might be dropped due to the applied rules. When this happens during the `dev_queue_xmit()` function call an error code will be returned. But if it happens later during the packet's traversing of the queue, no notification is possible.
3. Intermediate network structure nodes, e.g. switches, might become overburdened, especially when many communications occur simultaneously at the network. In that case they typically start dropping packets, which is unnoticeable to both the sending and receiving protocols.
4. Transmission errors are rare events in a closely interconnected Ethernet network, but they may appear. Every interface detecting an erroneous data frame by means of the checksum will silently drop it.
5. If at the remote host the Ethernet frames are received faster by the network device than the driver can hand them to the system for consumption, the receive ring will run full and the device will eventually start dropping following frames. This is imperceptible to both the sending and receiving protocols.
6. The target socket of the receiving host maintains a receive queue to temporarily store packets until the user-level process requests them using a reception function. If data is received faster than the application consumes it, this queue will run full and the typical action for this case is to drop following frames. This event, however, is noticeable by the target protocol.

The local and remote congestion issues could be addressed by the implementation of a flow control algorithm. The problem of possible transmission errors or other packet losses unnoticed by both the sending and receiving hosts, however, can not be addressed this way. But, since parallel processes need a reliable way to synchronize their processing or to interchange their computation results, the realized EDP protocol shall in the next step be extended by the means needed to provide reliable communication. Packet loss at the sender or receiver due to congestion will also be handled by this extension at first, although an additional flow control algorithm might be necessary to achieve the highest possible throughput.

Chapter 5: Implementation of an Ethernet Streaming Protocol

In this step the basic Ethernet datagram protocol shall be extended by the means needed to achieve a reliable communication. Thereby the low message latency as well as the independence from the underlying device driver shall be preserved. The extension will be implemented an independent protocol preserving EDP as a basic datagram protocol and for comparison reasons.

To achieve reliability a scheme of acknowledgements and retransmissions, similar to the one known from TCP [26], will be integrated. Thereby the receiver acknowledges the acceptance of data. The sender keeps a copy of any data transmitted until the corresponding acknowledgement has been received. Because both the transmitted data and the returned acknowledgement can be lost, the sender additionally maintains a timeout after which the data is considered lost and retransmitted. In this scheme it is necessary that at least individual data packets can be identified, e.g. by consecutive numbering. The receiver needs this information whenever new data arrives to detect whether any packets have been lost beforehand and to acknowledge the reception. The sender uses it to match the acknowledgements with the kept data.

The need for each receiver to know which data packet to expect next makes it necessary to introduce connections for two reasons: First, data packets from any other socket reachable over the network may arrive at an unconnected socket. Therefore it would have to store the information about the next expected packet for each communication partner and to be able to retrieve it efficiently. Second, any socket also needs an idea which data packet to accept as the first. Therefore the two peers of a connection need to start at predefined sequence numbers or to synchronize their initial sequence numbers at connection setup.

These considerations lead to the implementation of a streaming protocol providing connection-oriented and reliable data transport and communication multiplexing using ports.

5.1 The ESP Packet Structure

As shown in *Figure 5.1.1* the Ethernet Streaming Protocol (ESP) introduces additional fields to carry the sequence number of the packet and an acknowledgement and for flags needed to manage the state of an ESP connection.

The source and destination port fields again are used to multiplex several communications lead via a single interface, while the length field is used to detect and trim packets padded by the underlying Ethernet protocol. The flags field can contain a combination of the following flags:

- SYN: The remote host requests the synchronization (initiation) of a connection.
- ACK: The remote host acknowledges the reception of data packets.
- FIN: The remote host closes his side of the connection, i.e. stops sending data.
- RST: The remote host detected an unrecoverable irregularity in the communication and requests the connection to be reset.

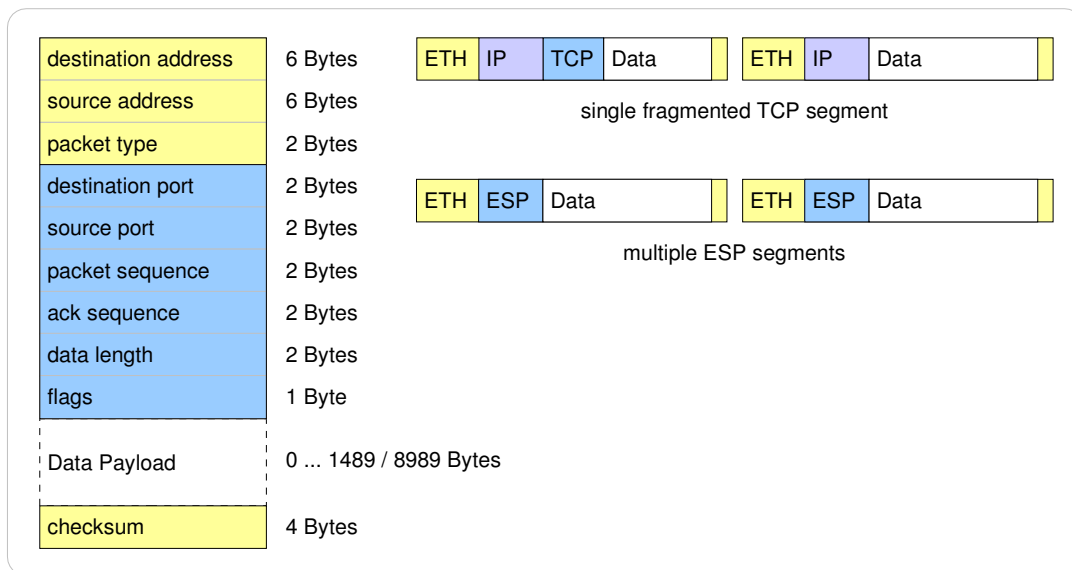


Figure 5.1.1: The ESP Packet Structure

With 11 bytes the ESP header is still only about half as large as the minimal IP [22] or TCP [26] header, both consuming 20 bytes. Thus, assuming an MTU of 1500 bytes, ESP would actually transmit 12 bytes to send a one-byte message compared to 41 bytes sent by TCP. For a 64 kB message ESP would actually transmit 66,031 bytes compared to 66,456 bytes sent by TCP. Summarized, this means a reduction of the amount of transferred data of about 71%, respectively 0.6%.

5.2 Implementation of the Ethernet Streaming Protocol

5.2.1 Registering the ESP Protocol

The `sock_enet_create()` function has already been registered with the kernel for the creation of sockets for the `PF_ENET` protocol family during the implementation of the datagram protocol. It is now extended to invoke the protocol-specific function `sock_esp_create()` whenever the user requests the creation of sockets of type `SOCK_STREAM`.

To be able to receive data ESP registers an own packet reception hook for the single packet type identified by the `ETH_P_ESP` constant. Its value can easily be adjusted in the associated header file under the same restrictions as explained in section 4.2.1.

5.2.2 ESP Socket States

While a connection passes through several states during its lifetime, e.g. from unconnected via connecting, established, closing to closed, the sockets forming the endpoints of the connection also move through various states as shown in *Figure 5.2.1*. The ESP state diagram represents a simplified TCP state diagram, basically reduced by the possibility of cross-connects and the *Time Wait* state. The processing of a single socket significantly depends on the state it is currently in.

As the diagram shows, the state of a socket can change in response to socket calls performed by the user-space process as well as asynchronously by the reception of data

packets containing certain connection management flags. If the socket state changes while a user space process is performing a socket call, the state diagram might be violated. And because state changes might include additional changes to the socket, unexpected state changes might lead to unpredictable results. Therefore the socket needs to be protected from changing its state while a user-space process performs a socket call.

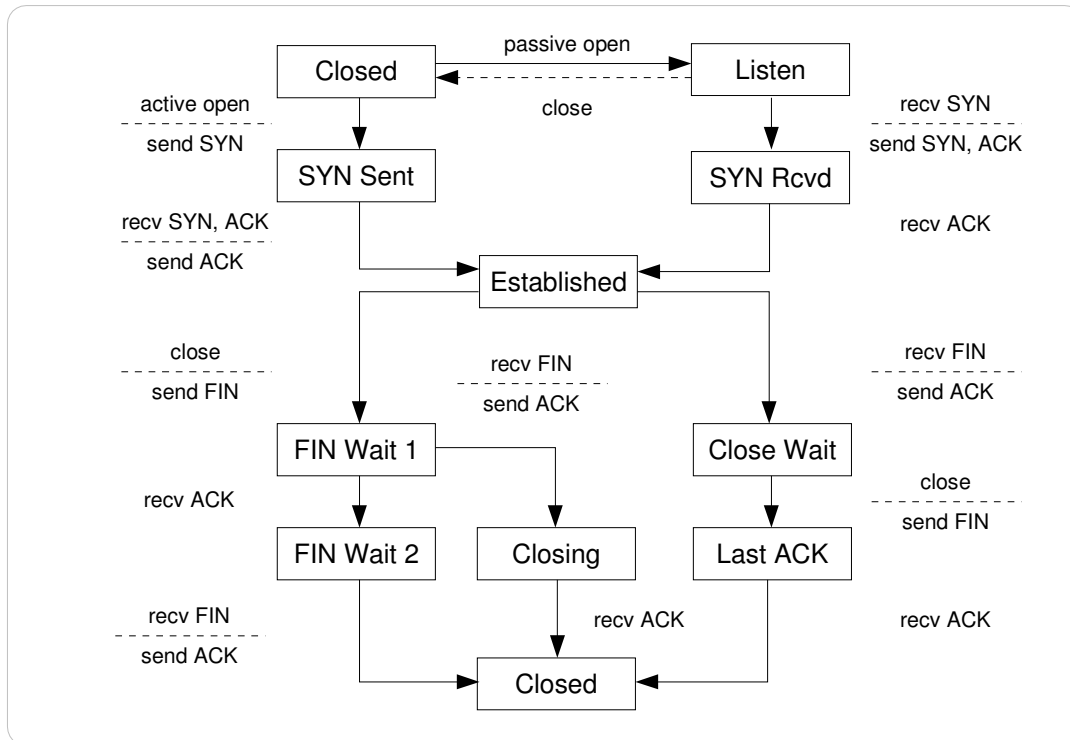


Figure 5.2.1: The ESP Socket States

As described in section 2.2.2 this is achieved by locking the socket using `lock_sock()` in user contexts, i.e. in the protocol interface functions, and `bh_lock_sock()` during interrupt processing, i.e. during asynchronous packet reception. The interrupt handler additionally uses the `sock_owned_by_user()` function to determine whether any user-space process currently executes a socket call on the socket. In this case received data packets are not processed immediately, but rather temporarily stored into the socket's backlog queue to prevent them from causing socket state changes. When the socket call ends, the socket lock will be released using `release_sock()` that additionally processes the packets kept in the backlog queue.

Locking the whole socket additionally removes the necessity of single locks for the several queues maintained by it to store data buffers or connection requests.

5.2.3 Socket Management

Each ESP socket can be in two out of three hashtables maintained by the ESP protocol. First, each socket bound to a local port and/or interface is inserted into the hashtable of bound sockets. Contrary to EDP this table is only used to determine the allocated or free ports. Second, sockets can be in one of the established or listening hash tables, depending on their states. This segregation is necessary, because listening sockets can be identified just by the local port and interface they have bound to, while target sockets

of an established connection are identified by the full 4-tuple of remote and local addresses and ports. Therefore two different hash functions must be applied to regard this difference. Both of them use the XOR operation to aggregate and fold all necessary values to a single 8 bit number. This way it is irrelevant whether they were provided in network or host byte order and the otherwise necessary byte order conversion can be omitted, as long as the hashtable does not exceed a size of 256 buckets.

5.2.4 Initiating a Connection

Before any data can be transmitted between two sockets, a connection has to be established to synchronize their sequence numbers. First, the passive end of the connection must be opened using the `listen()` socket call. The ESP listen handler is implemented by the `sock_esp_listen()` function. It checks whether the socket is in an appropriate state and tries to automatically bind a local port using the `esp_get_port()` function, if the socket is not already bound. If the socket isn't already in listening mode, it is inserted into the listening hashtable and set to the *Listen* state. Afterwards, the maximum backlog queue size for pending connection requests given by the user is entered into the socket's `sk_max_ack_backlog` field. The socket is now ready to receive connection synchronization requests.

When the passive end of a connection has been set up, other sockets can be actively connected to it using the `connect()` call. ESP performs a TCP-like three-way connection handshake consisting of

1. an initial packet with the `SYN` flag set, sent by the host actively opening a connection to announce its initial sending sequence number
2. an answering packet with the `SYN` and `ACK` flags set, confirming that the connection request has been accepted by the passive peer and announcing its initial sequence sending number
3. a final packet with the `ACK` flag set, returned by the initiator of the connection to confirm the sequence number of the passive peer.

The function `sock_esp_connect()` implements the `connect` handler for the ESP protocol. If the socket is in a closed state, the function invokes `esp_connect()`, which in turn does the following:

1. It determines the device descriptor from the socket address given by the user and checks, whether the device is up.
2. It automatically binds the socket to the device and the next free local port, if it is not bound already.
3. It calls `esp_sk_init_connect()` to store the address and port of the remote endpoint into the protocol-private socket data structure and set up the sequence numbers maintained there.
4. It invokes `esp_sk_hash_established()` to insert the socket into the established hashtable. This is necessary to enable the socket to receive data packets, e.g. the `SYN-ACK` sent by the peer to confirm the connection request.
5. It uses `esp_send_syn()` to build and send the initial `SYN` packet and changes the socket's state to *SYN Sent*.

In blocking mode the `esp_connect_wait()` function is invoked next to suspend the current process until the connection is established, i.e. the three-way handshake has been completed, or the socket's sending timeout has expired. On success the socket will now be in the *Established* state and connected to a remote socket and thus ready for data transmission.

Non-blocking sockets will just return the `EINPROGRESS` pseudo error code and leave it to the application to wait for the connection to complete, e.g. by polling the socket using the `select()` function.

5.2.5 Accepting a Connection

Whenever a listening socket receives a packet with the `SYN` flag set, it first checks, whether there is still space for a new connection request in its backlog queue. If so, it invokes the function `esp_sk_spawn_child()` to spawn a child socket representing the local endpoint of the new connection. This function allocates and initializes a new combined protocol-layer socket object from the protocol-private slab cache (cf. section 2.2.2) and sets it to the *SYN Rcvd* state. The new socket is enqueued to the `syn` queue, i.e. the queue of pending unfinished connection requests of the listening parent socket. This is necessary to be able to destroy the child socket structure, when the listening parent socket is closed before the connection has been finished or the child has been accepted by a user level process. The new socket is then inserted into the hashtable of bound sockets using the `esp_sk_hash_bound()` function, that also enters the bound local port and the index and address of the network interface the `SYN` packet has been received on into the socket structure. Afterwards, the remote address, port and the sequence numbers to use with the socket are initialized from the protocol headers of the `SYN` packet using `esp_sk_init_connect()`. Finally, the socket is inserted into the established hashtable to become able to receive subsequent packets. If the creation of the new child socket succeeded, the listening socket will invoke `esp_send_syn()` on it to respond to the connection request with a packet carrying the `SYN` and `ACK` flags.

The remaining part of the connection setup is handled by the child socket itself: When it receives the final `ACK` of the handshake it changes to the *Established* state and moves from the `syn` queue of its listening parent to the parent's queue of established connections acceptable by a user-space process.

An application can accept a pending connection request by performing an `accept()` call on a listening socket. ESP handles this call by means of the `sock_esp_accept()` function. It first checks, whether the socket it has been invoked on is in the *Listen* state and whether pending connection requests are waiting in the socket's accept queue. If the queue is empty and the function has been invoked in blocking mode, it will suspend the current process until a connection request has arrived or the socket's receive timeout has expired. In non-blocking mode the pseudo error code `EAGAIN` will be returned.

If there are connection requests pending, the function will fetch the first of them and invoke `sock_graft()` to link it to the BSD socket object pre-allocated by the kernel and provided to the accept handler as a parameter. The kernel will return the descriptor of the file associated to the BSD socket to the user-space application for use as socket handle.

5.2.6 Suspending the Current Process

There exist several ways for applications to influence the blocking behaviour of socket calls. A new socket is initially set to blocking mode, i.e. socket calls will suspend the current process, if they can't serve the request immediately. Each socket maintains two separate timeouts for sending and receiving operations, after which a suspended process is reawakened. Both of them can be adjusted by invoking the `setsockopt()` call on the socket file descriptor providing the `SOL_SOCKET` socket level and the `SO_SNDTIMEO`, respectively `SO_RCVTIMEO` option name.

Single socket calls can be requested to operate non-blocking by specifying the `MSG_DONTWAIT` flag. Additionally, it is possible to generally switch the socket to non-blocking mode by setting the `O_NONBLOCK` flag on its descriptor using `fcntl()`. The socket calls will return the `EAGAIN` pseudo error code, if a request can not be served right away and the socket is set to non-blocking mode or if the appropriate timeout expired.

A socket's interface function at kernel level can check the provided flags to determine, whether non-blocking operation is requested. The kernel will hand the `MSG_DONTWAIT` flag to all but two of them in this case. The `accept()` and `connect()` socket calls at user level don't take a flags parameter, and so the kernel will hand the `O_NONBLOCK` flag of the socket's file to the associated kernel level protocol interface functions to request non-blocking behaviour.

The interface functions can use the `sock_sndtimeo()` or `sock_rcvtimeo()` functions to determine a socket's timeout values. Additionally the kernel provides some default functions, like `sk_stream_wait_connect()` or `sk_stream_wait_memory()`, to suspend the current process in certain default cases. If those default functions cannot be applied, the suspending must be done manually. For that purpose a new wait queue entry must be defined and initialized with the descriptor of the current process with the `DEFINE_WAIT()` macro. Then it has to be inserted into the socket's wait queue `sk->sk_sleep` with one of the `prepare_to_wait()` or `prepare_to_wait_exclusive()` functions. The difference is that exclusively waiting processes can be awoken one-by-one, e.g. to avoid the 'thundering herd' effect, while the others are always awoken together. Afterwards the preconditions of the waiting are checked, e.g. whether the socket is in the correct state, whether errors are pending or whether the timeout has already expired. Then the `sk_wait_event()` macro can be invoked providing a reference to the socket, the timeout value and a condition to wait for.

The macro will release the socket's lock to process the socket's backlog packet queue and allow for unrestricted interrupt level processing during the waiting period, including possible socket state changes. If the given condition is not yet met, it will suspend the current process by a call to `schedule_timeout()`. When this call returns, i.e. the process has been re-awoken, the macro again checks the condition, re-locks the socket and returns the checking result.

If the condition that has been waited for is not yet met, the typical action is to re-start the procedure, inserting the wait queue entry into the socket's wait queue. Otherwise the wait queue entry has to be removed from the queue by a call to `finish_wait()`.

5.2.7 Sending User Data

The `sock_esp_sendmsg()` function implements the ESP handler for user data transmission. It first checks, whether the socket it has been invoked on is part of an established connection. If not, it tries to wait for a pending connection to finish using `sk_stream_wait_connect()`. This kernel socket function, like some others, uses the socket state constants defined by the TCP protocol. Therefore it is necessary to use the same constants for a new protocol, if the predefined kernel functions shall be used. Next it checks, whether the socket has already been shut down for sending, in which case an error is returned. Afterwards it determines the descriptor of the device the socket has bound to and checks, whether it is up.

The sockets of the new ESP protocol provide stream-oriented connections. Therefore they must be able to handle transmission requests of arbitrary message sizes. To minimize the system call overhead, the `sock_esp_sendmsg()` function is able to process message sizes larger than the MTU of the protocol in a single call by splitting the message to MTU-sized chunks. To allocate and initialize a socket buffer for a message chunk the function `esp_skb_alloc_sk()` is called, which in turn does the following:

1. It first invokes the `esp_skb_alloc()` function, which in turn
 - 1.a. allocates the socket buffer using `alloc_skb()`
 - 1.b. creates the Ethernet protocol header using the `dev->hard_header()` function of the descriptor of the output device
2. It creates the ESP packet header from the parameters handed to the function and the port and current sequence numbers of the connection stored with the socket.

Next the `sock_esp_sendmsg()` function calls `memcpy_fromiovec()` to copy the next part of the message into the socket buffer, which is then enqueued to the socket's retransmission queue. As the new ESP protocol does not yet implement any flow control mechanism, a clone of each prepared socket buffer is immediately handed to `esp_skb_xmit()` for sending. The cloning is necessary, because the socket buffer has been enqueued, but the reference provided to the transmission function will be released after the transmission has ended.

Whenever there is no send buffer space available at the socket to allocate the socket buffer for another piece of the message, the function will suspend the current process using `sk_stream_wait_memory()`.

5.2.8 Managing Retransmissions

Two kinds of packets can be sent by the ESP protocol: Packets containing user data, which originate from application sending requests, and packets containing only flags, which are generated by the protocol itself in response to certain events to manage the connection. User data packets as well as the management packets necessary to setup or gracefully shutdown a connection, i.e. packets containing the `SYN` or `FIN` flags, need to be transported reliably. Therefore the socket buffers created to send those packets are not discarded after the transmission, but rather kept in a per-socket retransmission queue. The `esp_skb_enqueue_rtx()` function is used to enqueue the packets. It additionally starts a timer for the socket's retransmission timeout.

The retransmission timeout is determined dynamically from the time interval between the transmission of a packet and the reception of a matching acknowledgement. For that purpose the amount of transmissions and the timestamp of the last one is stored into the protocol-private area of each socket buffer. When the matching acknowledgement is received the current round trip time (rtt) is calculated from the difference of the current time and the transmission time of the socket buffer. But this calculation is only performed for packets that have been sent once. Otherwise it could not be told whether the acknowledgement refers to the original transmission or one of the retransmissions of the packet. Following, a smoothed round trip time (srtt) is derived from the round trip time value with the formula $srtt = 7/8 srtt + 1/8 rtt$. Finally, the retransmission timeout (rto) is calculated to be twice the smoothed round trip time to ensure, that it does not expire even if it takes a little longer until the next acknowledgement arrives. It is, however, limited by the values of the `ESP_MIN_RTO` and `ESP_MAX_RTO` constants.

Whenever the retransmission timer expires, the registered `esp_rtx_timer()` callback function will be executed. It traverses the retransmission queue and compares the individual timeout values of the queued socket buffers with the current time. A clone of each socket buffer whose timeout has expired is handed `esp_skb_xmit()` for retransmission. The cloning again is necessary, because the original socket buffer will stay in the queue until it has been acknowledged by the peer.

The current implementation allows for manipulation of the retransmission timeout and behaviour in two ways: First, it is possible to limit the amount of retransmission attempts for a single socket buffer by adjusting the `ESP_MAX_RETRANSMITS` constant to the desired value. The connection is considered dead whenever this value is exceeded and a write error is reported to the user-space process. Second the timeout can be doubled after each retransmission round to reduce the eagerness of the protocol with the `ESP_DOUBLE_RTO` constant. If additionally the `ESP_TIMEOUT_CONN` constant has been set to a non-zero value, the connection again will be considered dead whenever the timeout value exceeded the configured maximum and a write error is reported to the user-space process.

5.2.9 Receiving ESP Data Packets

The data reception of ESP is divided into two stages: The packet reception hook function `esp_rcv_hook()` registered with the kernel and the packet processing function `esp_rcv_packet()`.

Initially the reception hook function performs the same actions as the EDP hook:

1. It checks whether the packet is acceptable for the local host.
2. It invokes `skb_share_check()` to get a private clone of the socket buffer, if it is shared with another handler.
3. It invokes `pskb_may_pull()` to check whether the socket buffer contains at least a complete ESP packet header and if so, adjusts the socket buffer's data pointer to the beginning of the data using `__pskb_pull()`.

Next the function needs to determine the target socket for the received packet. For that purpose the function `esp_sk_lookup()` is called, which in turn invokes

`esp_sk_lookup_established()` first to search a socket with an established connection to the origin of the packet. If no connected socket was found, next the `esp_sk_lookup_listen()` function is invoked to look for a socket listening at the destination port of the packet. If no target socket was found, the reception hook will discard the received packet after having informed its originator with a packet carrying the RST flag.

If the packet is acceptable and the target socket has been determined, the reception hook function next locks the socket using `bh_lock_sock()`. It then determines whether the socket is currently in use by a user-space process performing a socket call using the `sock_owned_by_user()` function. If not, the packet is directly handed to the packet processing function `esp_rcv_packet()`. Otherwise the socket buffer is stored temporarily at the socket's packet backlog queue using `sk_add_backlog()` to prevent it from causing socket state changes. When the socket call ends `release_sock()` will be invoked to release the socket's lock. It will additionally process the backlog queue and hand each socket buffer contained therein to the `sk->sk_backlog_rcv()` function of the socket, that is also set to `esp_rcv_packet()` for ESP sockets upon their creation.

The ESP data reception hook function unlocks the socket by a call to `bh_unlock_sock()` before it ends.

5.2.10 Processing Received Packets

The `esp_rcv_packet()` function is the main processing function for received packets. When it is invoked the socket has already been locked and it is ensured not to be in use by a socket call, so it is safe to change the socket's state from within the function.

Basically, the processing function performs any socket state transitions triggered by incoming packets, initiates the creation of new sockets due to synchronization requests on listening sockets or enqueues data to the receive queue of connected ones. Thereby it performs the following basic steps depending on the socket's state:

1. In synchronized states it checks, whether the packet contains an acceptable sequence number. For that purpose the `seq_before()` and `seq_after()` functions are used, that correctly handle the wrap-around of the unsigned number space. If the packet's sequence number lies before the next expected one, its originator is immediately informed about the lost synchronization with an acknowledgement carrying the next expected sequence number.
2. The flags field of the packet's ESP header is checked for the presence of single flags. The main packet processing function invokes `esp_process_ack()`, `esp_process_fin()` or `esp_process_rst()` to handle the presence of the corresponding flags, which will perform the necessary actions and/or socket state changes. Sockets in the *Listen* state will spawn new child sockets as described in section 5.2.5 upon the reception of packets with the *SYN* flag set.
3. If the socket's current state allows for the reception of data, socket buffers containing data are enqueued to the socket's receive queue. After having enqueued a new data packet each socket must update, i.e. increase, the next expected sequence number to indicate the progress of the communication.

Two basic approaches can be followed whenever in synchronized socket states packets are received that carry a sequence number after the next expected one. First, they can be silently discarded. The retransmission mechanism will send them again after the missed packets. Second, the socket could store them in an additional queue. In this case the out-of-sequence packets kept in the additional queue must also be charged to the socket's receive buffer to prevent it from consuming all of the system's memory. Furthermore the out-of-sequence queue must leave enough free buffer space for at least one in-sequence packet. Otherwise the socket would continue to drop the next expected packet due to its lack of receive buffer space, thus preventing the communication from proceeding.

But leaving receive buffer space for a fix number n of in-sequence packet always bears the possibility that $k > n$ packets are missed. This would lead to the necessity of $r = \left\lceil \frac{k}{n} \right\rceil$ additional retransmission rounds or an additional delay of r times the retransmission timeout. Therefore it is better to leave receive buffer space for as much packets as needed to fill the hole between the last in-sequence and the first out-of-sequence packets.

Although such an out-of-order queue has been implemented for the ESP protocol, tests showed that it was barely used: Out of over 500.000 packets not even 200 entered the queue. Therefore it has been deactivated to avoid the additional processing overhead in the main communication path.

5.2.11 Acknowledging Received Packets

To keep the ESP packet headers as small as possible they initially contained only a single sequence number field. Depending on the flags it was used for both purposes: to identify a particular data packet or to notify the sender about the reception of one. Additionally the initial protocol variant expected a selective acknowledgement of every transmitted packet. This procedure, however, introduces an acknowledgement processing overhead that leads to a significant reduction of the maximum throughput of the protocol.

Therefore the current implementation of ESP, like TCP, sends only a single acknowledgement containing the next expected sequence number to confirm the reception of all data packets with sequence numbers before that one. Furthermore, an additional field for the acknowledgement number has been added to the ESP packet header. This opens the possibility to include acknowledgements in data packets reducing the minimum amount of additionally needed ACK packets to zero in the case of dialogue communication, i.e. alternating sending and receiving.

To keep track of pending ACKs each socket stores the acknowledgement number last sent. Whenever this number differs from the next expected sequence number, acknowledgements are pending. Each packet sending function checks for pending ACKs using the `ESP_ACK_PENDING` macro and if so, includes the `ACK` flag in the packet header. This way unnecessary processing overhead is avoided at the receiver that would result from the `ACK` flag being set in every data packet.

An additional ACK timeout is maintained at each socket for the case that no data has to be sent for a certain amount of time. Therefore `esp_dack_schedule()` is called

whenever new in-sequence data has been enqueued to the socket. It first checks whether the current amount of pending ACKs exceeds the value of the `ESP_MAX_ACK_PENDING` constant. If not, the ACK timer is activated. Otherwise or when the timeout expires a separate acknowledgement packet is transmitted.

After having sent an acknowledgement, either separately or included into another packet, the pending ACKs are cleared and the ACK timer is stopped using the `esp_dack_clear()` function.

As described in section 5.2.8 an ESP socket keeps a copy of each transmitted data packet in its retransmission queue. Additionally it stores the sequence number of the first packet transmitted, but not yet acknowledged. Whenever an acknowledgement arrives the main packet processing function will call `esp_process_ack()` to handle it. This function first checks, whether the acknowledgement number of the packet is acceptable, i.e. lies between the first unacknowledged and the next-to-send sequence numbers. If so, the function next traverses the retransmission queue and removes each acknowledged packet. Afterwards the stored sequence number of the first unacknowledged packet is updated and the retransmission timer is stopped, if the queue has been emptied, or restarted otherwise. Finally, all processes waiting on the socket for write buffer space are notified with a call to the socket's `sk->sk_write_space()` function.

5.3 ESP Micro-Benchmarks

5.3.1 Message Latency and Throughput

The following *Figures 5.3.1 - 5.3.4* show the message latency and throughput values of the new ESP protocol as a function of the message size. They have been measured with the `netgauge` network benchmarking tool on test systems of both clusters by performing 2.500 test runs for each data size and calculating the minimum or maximum values.

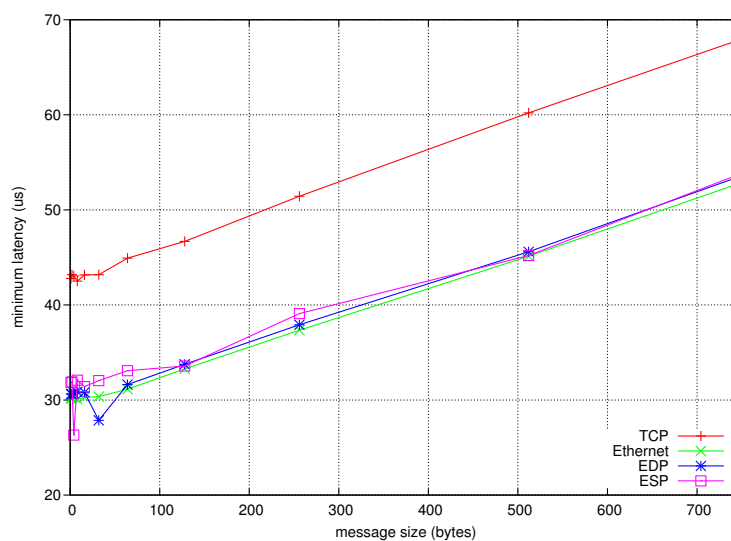


Figure 5.3.1: ESP Message Latency on Cluster-1

The benchmarks on the first test cluster system show, that by keeping the implementation of the sending and receiving protocol handler functions as small as possible, ESP is

nearly able to achieve the low message latency values of the previously developed datagram protocol. In fact, it only introduces an additional latency of about 1.5 μ s. The throughput values of ESP for large messages, however, clearly stay behind those of EDP by about 136 Mbit/s or 17 %.

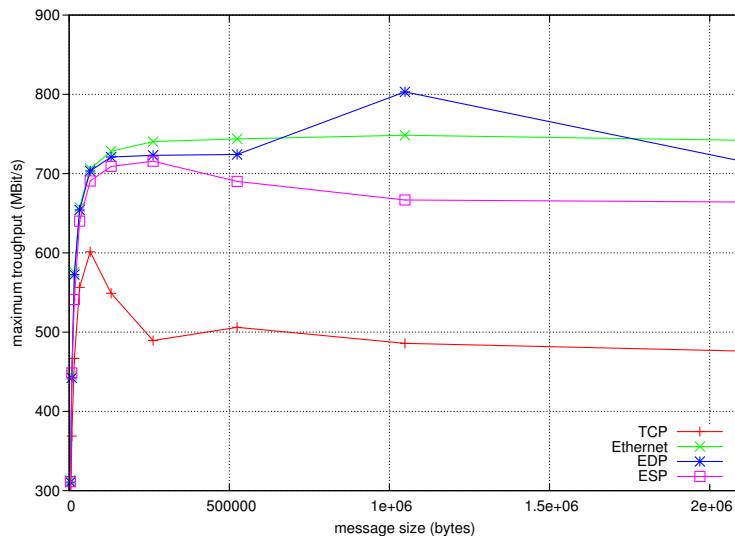


Figure 5.3.2: ESP Message Throughput on Cluster-1

On the second test cluster a similar result could be achieved for the message latency. The throughput values of ESP, however, exceed those of the datagram protocol for large message sizes by 3 %. This might be due to the fact that ESP only requires a single system call for large messages and fragments them in the kernel, while EDP only takes data sizes of the protocol MTU, i.e. needs several system calls.

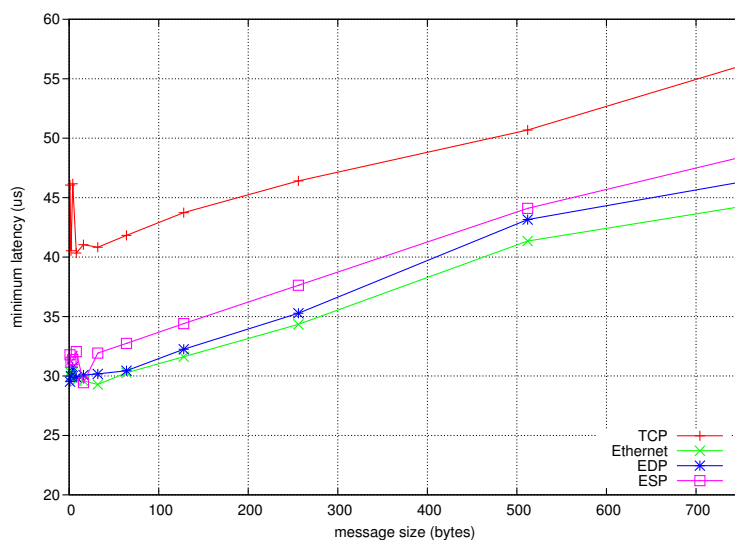


Figure 5.3.3: ESP Message Latency on Cluster-2

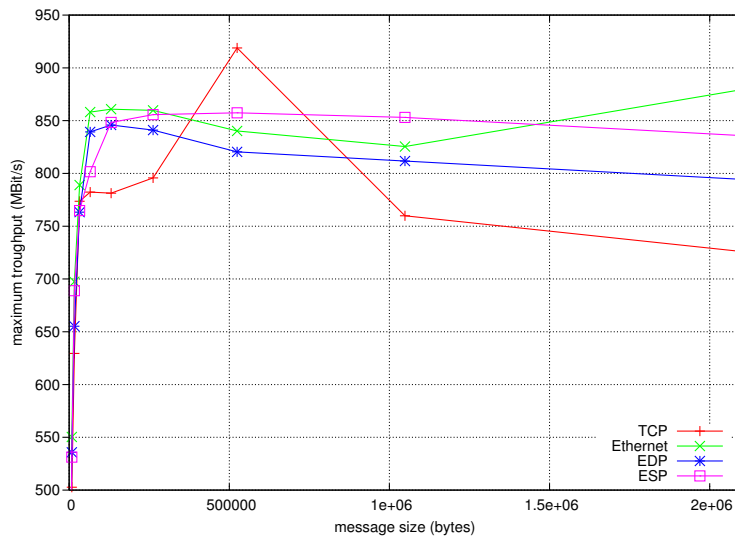


Figure 5.3.4: ESP Message Throughput on Cluster-2

5.3.2 Influence of Acknowledgement Processing

As stated before, the processing of the acknowledgements returned by the remote connection peer can influence the performance of the protocol. Thereby the ACKs don't necessarily have to arrive separately, as the remote interface does also maintain a packet queue and might decide to send consecutive ACKs in a small bulk. For each received single or group of ACKs, however, the current processing of a host is interrupted. And because ACKs are sent in response to data being transmitted, this holds especially true, if it is currently engaged sending a large amount of data. Therefore the maximum throughput for large messages can be affected, as it is shown in Figure 5.3.5.

There the message latency and throughput values are pictured with the maximum allowed amount of pending ACKs set to several values from 1 through 15. With acknowledgements sent every 1 or 5 packets the throughput reduced by about 100 Mbit/s.

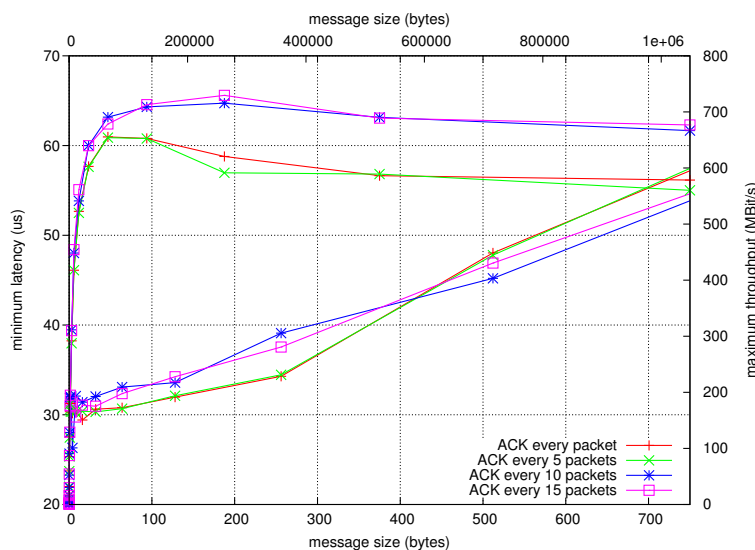


Figure 5.3.5: ESP ACK Processing Influence on Message Latency and Throughput on Cluster-1

5.3.3 Average Throughput, Influence of the Socket Buffer Sizes

Especially the average throughput of ESP for large message sizes stays clearly behind the TCP values. Since ESP does not implement flow control it must rely on the retransmissions whenever a packet was lost due to error or congestion. Because the current Linux kernel does only provide coarse-grained timers based on jiffies, i.e. with a resolution of about 10 ms, any retransmission largely delays the flow of the communication.

The following *Figures 5.3.6 - 5.3.7* show the latency and throughput values of ESP for various send and receive socket buffer sizes. It is clearly to be seen, that for a small receive buffer size of 64 kB even the maximum throughput value drops below that of TCP. The average throughput, however, always stays below that of TCP and can only little be influenced by any socket buffer changes.

To increase the throughput, flow control should be implemented in a future extension to avoid retransmissions as far as possible.

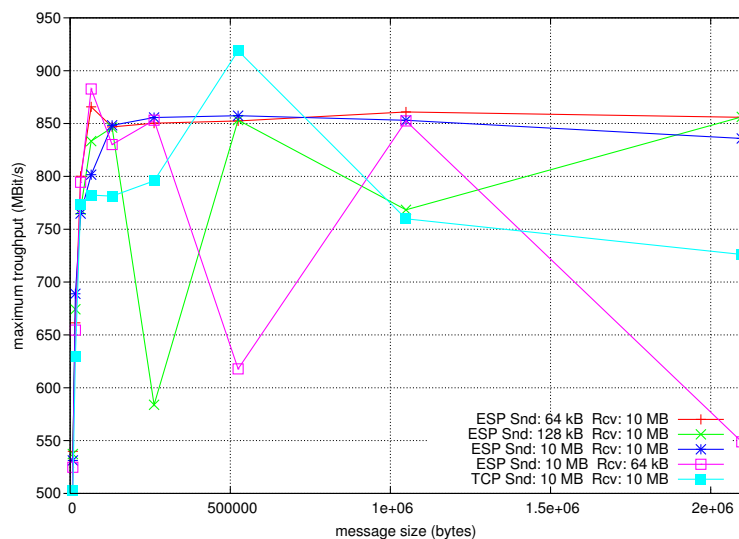


Figure 5.3.6: Socket Buffer Size Influence on the ESP Maximum Message Throughput on Cluster-2

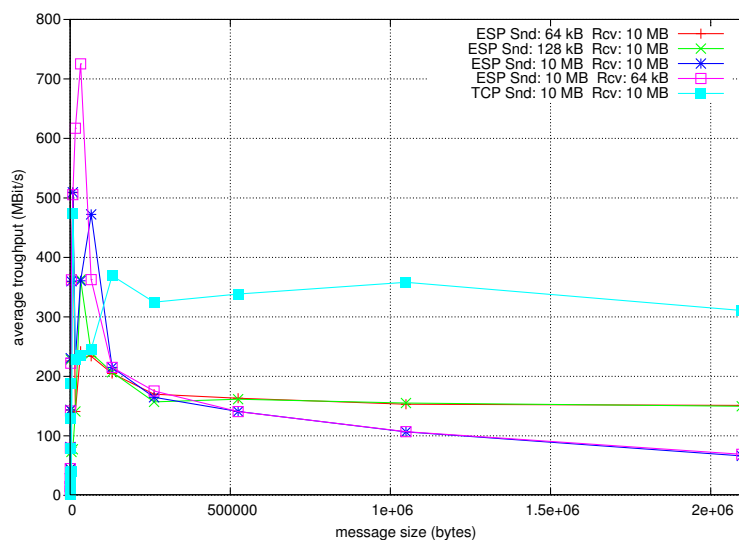


Figure 5.3.7: Socket Buffer Size Influence on the ESP Average Message Throughput on Cluster-2

Chapter 6: Open MPI v1.0

Open MPI [19, 27] is a new open source implementation of the MPI-2 [28] standard. It evolved from the experiences gained with various other MPI implementations and tries to combine the benefits of each one. The following list states some of the main goals and features of Open MPI:

- full MPI-2 standards conformance
- support for many operating systems
- production quality software
- tunable by installers and end-users
- portable and maintainable
- support data and network heterogeneity
- single library supports all networks
- component-based design, documented APIs

Open MPI is based upon a Modular Component Architecture (MCA) providing the means needed to easily add new functionality. It provides a stable platform for third-party research and enables an easy run-time configuration by adding modules that provide the needed functionality.

6.1 The Modular Component Architecture

As shown in *Figure 6.1.1* the modular component architecture is the backbone of the Open MPI implementation. It manages the single component frameworks and provides basic services to them, like the discovery and loading of components or the obtainment of run-time parameters and their passing to single modules.

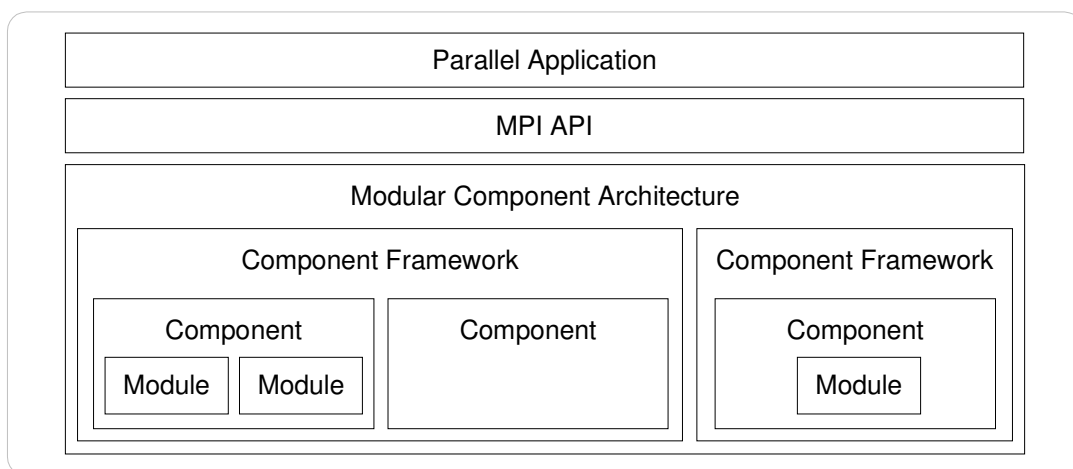


Figure 6.1.1: The Modular Component Architecture

Each of the *component frameworks* has to take care of a certain task, e.g. collective operations or network communication. For that purpose each framework contains several *components*, that are able to accomplish the task, probably in different ways. The binary transmission layer (BTL) for instance contains several components allowing for communication over different network architectures. The framework chooses one of them, e.g. according to the present system hardware, precedence rules or run-time configuration, loads and initializes it. Upon initialization the component instantiates one or more modules as required to fulfil the particular task. Thus a communication component could create one module per available network interface.

Open MPI uses an own light-weight object oriented programming scheme to efficiently manage its modularity. It uses C-language data structures containing data fields and function pointers as objects and special macros to emulate the object creation and destruction semantics. This way also single inheritance and object memory management based on reference counting has been implemented.

6.2 The TCP BTL Component

The components of the Binary Transmission Layer (BTL) are the means for Open MPI to abstract from the underlying communication protocol and/or hardware. The interface functions of a BTL component are invoked by components of other frameworks, namely the Binary Management Layer (BML), the Point-to-point Management Layer (PML) or the collectives framework (COLL) for low-level transmission of opaque data. Therefore BTL components need no particular knowledge of the MPI semantics.

6.2.1 The Lifecycle of the TCP BTL Component

During the `MPI_INIT()` call the MCA initializes the component frameworks. Those in turn will search all available components, that may have been linked statically into the MPI library or be shared libraries residing in one of the configured component directories. The MCA identifies loadable components by a strict file naming scheme indicating the framework it belongs to as well as its name. Next, the descriptor of the component, an object derived from `mca_base_component_t`, is examined and its `mca_open_component()` function is called. The open function of the TCP BTL component registers several run-time parameters with the MCA framework, which hands their values to the component, if they were configured. Otherwise default values are used.

In the next step the TCP component is initialized. It creates a single module for each of the system's network interfaces, except for those excluded by configuration, and a single listening socket used later to accept connections from remote processes. Afterwards it announces the addresses of the local interfaces and the port bound by the listening socket to the remote processes by means of the module exchange. Upon success the initialization function returns a list of the created modules. Upon failure a component can return a `NULL` list to indicate that its services are not available, e.g. when necessary hardware is not present in a system.

The Point-to-point Management Layer (PML) maintains a global registry where each component can store information it wants to share with remote components of the same type, e.g. information needed to connect to the process. Between the component initialization and the following selection phases the stored values will be exchanged with all other processes by the out-of-band module exchange.

After the components have been initialized, the MCA will select one or more of them to be used. For BTL components this happens by adding remote processes. Thereby the `mca_btl_base_add_proc_fn_t()` of each module is called with a list of remote process identifiers. It is the task of this function to decide whether the current module can reach the remote process. The TCP modules do this by comparing the subnet bits of the local and remote addresses. If no remote address in the same subnet as the local interface is found, the first available remote address is used. Since TCP/IP

supports routing, this might work as well. The function will return a bit mask to the MCA with the bits corresponding to all reachable processes set. The MCA then decides by an internal priority ranking of the BTL components and the discovered reachability of remote processes, which components should be used for data transmission to single remote processes.

When the MPI process shuts down, the close function of each component will be called to enable the components to cleanup and release all allocated resources.

6.2.2 Managing Remote Processes

The TCP BTL component maintains a hashtable of descriptors of all known remote processes, which is shared among all instantiated modules. The network addresses exported by a remote process and obtained through the module exchange are stored in an array together with each process descriptor.

Upon component selection, i.e. when the TCP modules check the reachability of the remote addresses, they create additional endpoint descriptors containing a reference to the remote process and the address by which it is reachable. The endpoints are returned to the upper layers of Open MPI. They will be provided as parameter to data transmission functions to identify the destination.

6.2.3 Data Transmission

The TCP BTL component supports lazy connecting to minimize the amount of file descriptors needed. At startup time only a single socket will be created listening on all available interfaces and connections to remote processes are not established until the first data transmission is requested.

Data transmission is nearly completely handled by TCP endpoints. Upon the first transmission request the necessary network socket is created, associated to the endpoint and the connection to the remote process is initiated. The TCP BTL component uses non-blocking sockets. To be notified of socket events an event object is registered with the event library of Open MPI. It contains the socket's descriptor and pointers to the endpoint's receive and send handler functions to be invoked whenever an associated event occurs.

When the connection is established, the data transmission service of a BTL can be used by several other components. Therefore additionally the target component needs to be addressed and a BTL needs to know how different components can be notified about received data. For that purpose each component that wants to use the basic transmission services registers with the BTL, providing a special tag value and its data reception callback function. The BTL in turn maintains a list of all registered callbacks. When data shall be transmitted, the upper layers of Open MPI additionally provide the endpoint identifying the target process and the tag of the target component to the transmission function of the BTL. This in turn prepends the data with a private header containing the data length and the provided tag before transmission. The receiving BTL inspects the tag and invokes the registered callback to hand the data to the target component.

6.3 Implementation of an Ethernet BTL Component

As the underlying ESP protocol provides stream sockets to the user-space just like TCP, the adaptations needed to port the existing TCP BTL atop ESP are minimal:

1. The `af_enet.h` header file needs to be included to get the needed data structures and constants.
2. All `socket()` calls must be changed to create `SOCK_STREAM` sockets from the new `PF_ENET` protocol family and the `EPPROTO_ESP` protocol.
3. All socket addresses must be converted to the new Ethernet socket address structure `struct sockaddr_en`. Assignments and comparisons of address variables have to be adapted to work with the 6 byte Ethernet address array. Upon creation each Ethernet BTL module must determine the hardware address of the interface it is bound to using `ioctl()` with the `SIOCGIFHWADDR` option name. Furthermore, the out-of-band module exchange must be updated to announce the Ethernet hardware addresses of all bound devices to remote processes.
4. Whenever a remote process is added the local output network interface has to be determined through which the new process can be reached, because the Ethernet protocol family does not implement routing.

The fourth point shall be explained a little more in detail. When a new process is added, its remote Ethernet address and port can be determined from the global registry managed by the out-of-band module exchange. But Ethernet addresses, unlike IP addresses, are *not* logically grouped. For the IP protocol a local network interface on the same subnet as the remote one can easily be determined by comparing the subnet bits of the remote address with the subnet bits of each local interface address. And even if no interface on the same subnet is found, the internet protocol might still be able to route the communication to the desired endpoint. But Ethernet addresses provide no means of logical grouping and Ethernet does not support any routing functionality.

Therefore it is necessary to determine a local interface on the same physical subnet as the remote one, i.e. connected to it directly by wire or via an intermediate level-1 or level-2 switching architecture. The current implementation of the Open MPI Ethernet BTL uses EDP and its sockets to perform this task. Each BTL module is bound to a single network interface. When a new process is added, EDP echo request control messages are sent to each of the addresses the process announced and that are not used by any other local Ethernet BTL, yet. The amount and timeout of the requests can be adjusted using the `btl_eth_echo_attempts` and `btl_eth_echo_timeout` command line parameters, respectively. If an echo reply is received from the remote host, the corresponding address is entered into the local endpoint data structure to be used for further communication. If the timeout expires before a reply was received, the currently checked remote address is considered unreachable by the current BTL module and the function tries the next one of the remote process. Furthermore, if none of the addresses announced by the remote process could be reached, the whole process is considered unreachable by the current BTL.

6.4 Micro-Benchmarks

6.4.1 Benchmarks of the Pallas MPI Benchmark Suite

The Pallas MPI Benchmark Suite (PMB) [29] has been developed to help in comparing the performance of several message passing libraries, computer platforms and used communication means. Its basic goals are

- to provide a concise set of benchmarks targeted at measuring the most important MPI functions
- to set forth a precise benchmark methodology
- to report bare timings instead of trying to interpret any results

The following Figures 6.4.1 - 6.4.4. show the results of a few selected benchmarks. First, the standard Ping-Pong test has been executed, that alternately sends and receives messages of increasing size. Afterwards several collective operations have been measured.

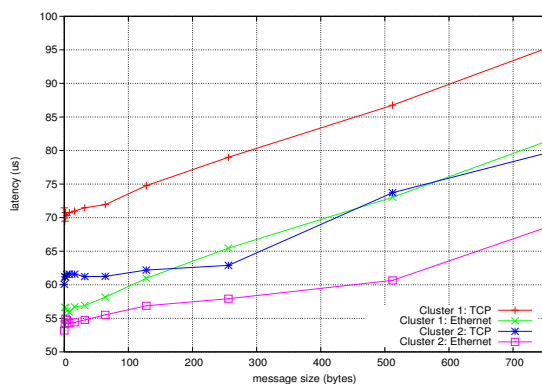


Figure 6.4.1: Results of the PMB Ping-Pong Benchmark

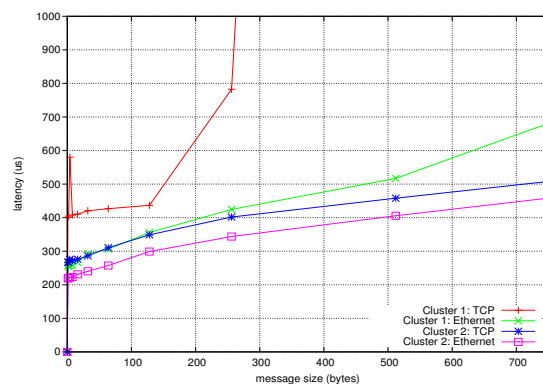


Figure 6.4.2: Results of the PMB Allgather Benchmark

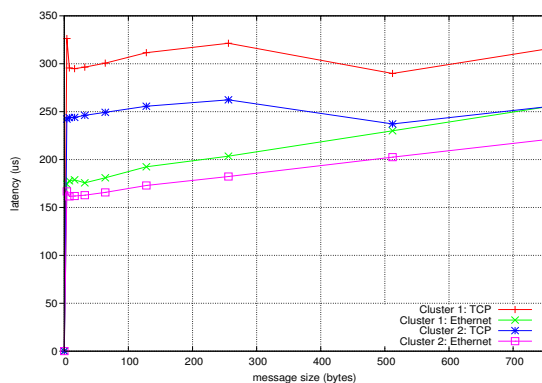


Figure 6.4.3: Results of the PMB Reduce Benchmark

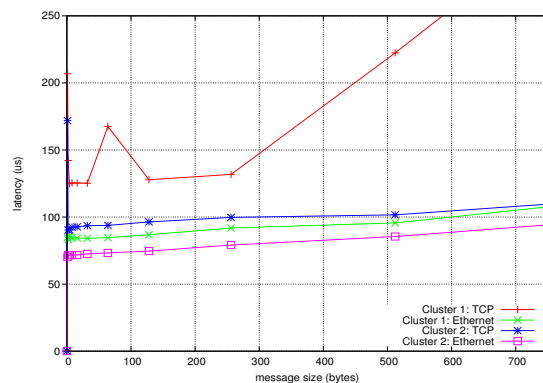


Figure 6.4.4: Results of the PMB Broadcast Benchmark

The benchmarks show, that the message latency of all operations could be significantly reduced compared to TCP by using the Ethernet BTL component and the underlying ESP protocol.

6.4.2 Communication Module Profiling

When a parallel application sends or receives data, several software layers are invoked, e.g. the MPI library, the involved network protocols, the network device interface of the kernel and the driver of the NIC. To obtain an overview on how much time is spent in the single layers, a simple parallel program has been used, transmitting one double value between two hosts and consisting of the following steps:

rank 0:

1. MPI-Barrier()
2. get_timestamp(t1)
3. MPI_Send()
4. get_timestamp(t2)
5. MPI_Recv()
6. get_timestamp(t3)

rank 1:

1. MPI-Barrier()
2. MPI_Recv()
3. MPI_Send()

Additionally timestamps have been taken in the ESP protocol interface functions and in the transmission and reception functions of an extended NIC device driver. The `rdtsc` operation of the Intel processor has been used to obtain precise values with a high resolution for the timestamps. The `cpu` clock cycles returned by the operation have then been divided by the clock frequency to convert them to microseconds. The measurement and the entailed output of the values, however, influenced the overall operation leading to larger latency value than those measured in the benchmarks before.

Figure 6.4.5 shows the times measured for the TCP module. The sending call time has been taken between the invocations of `MPI_Send()` and the hardware transmission function of the driver. There the message is handed to the NIC and considered to be on the network and the following value denotes the time needed afterwards to return from the `MPI_Send()` call. Afterwards the processor waits for the transmission idle time in the `MPI_Recv()` call. The reception time value denotes the time the message needs from the hardware reception function of the driver until it is returned from the `MPI_Recv()` function. This last value is quite large, e.g. compared to the time needed for the `MPI_Send()` call. Since the system call overhead is contained in both values, it indicates that Open MPI introduces a significant additional latency upon message reception.

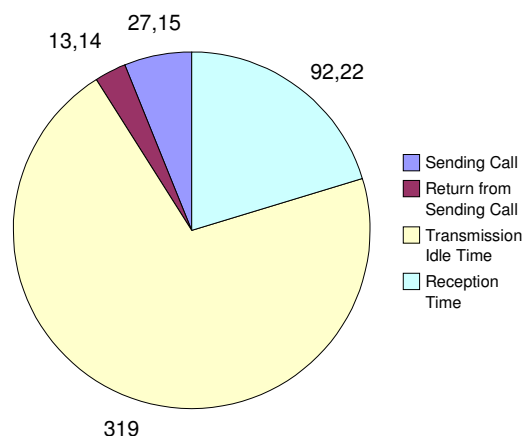


Figure 6.4.5: Profiling of an Open MPI Communication using TCP on Cluster-2

Afterwards the same test has been performed with the Ethernet module upon the ESP protocol. Here a few more timestamps were taken inside the ESP module to get some information about its performance.

The values again start with the invocation of the `MPI_Send()` function. The first value denotes the time needed until the transmission interface function of ESP is invoked, i.e. the time through Open MPI and an additional system call. Together with the next interval needed until the hardware transmission function of the driver is invoked it takes $19,5 \mu\text{s}$ compared to $27,15 \mu\text{s}$ of the TCP value. The following values denote the remaining parts of the ESP transmission and the `MPI_Send()` functions and the time the processor spends waiting in `MPI_Recv()`, respectively.

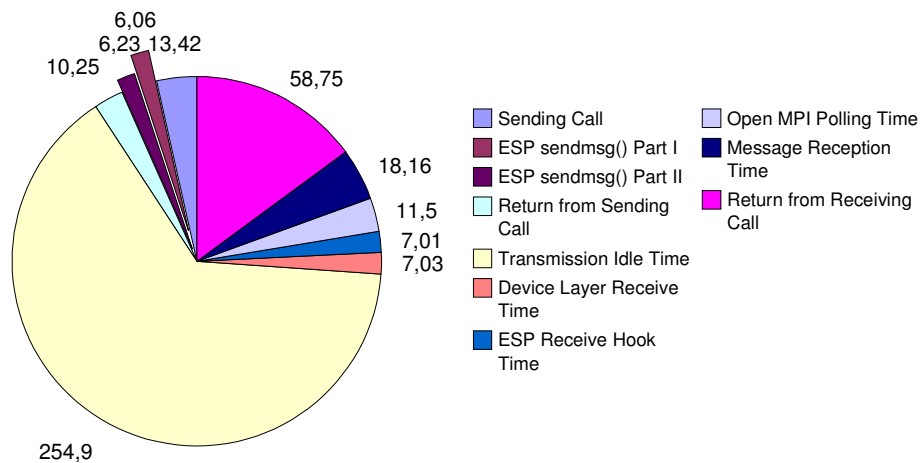


Figure 6.4.6: Profiling of an Open MPI Communication using ESP

The next two values denote the time the received message needs through the device interface layer and the interrupt data reception handler of ESP. The Open MPI polling time describes the interval between the enqueueing of the received message to the socket's receive queue and the reception request from the Open MPI library. The event library of Open MPI uses polling to query the socket for events, e.g. the reception of data. It is clearly to be seen that although this polling introduces an additional latency upon message reception, it is not its main cause.

Next, the message reception time denotes the time spent in the ESP data reception interface function and the last value shows the time needed to return from `MPI_Recv()`. The sum of the last five ESP values of $102,5 \mu\text{s}$ can again be compared to the last TCP value of about $92 \mu\text{s}$, but the additional $10 \mu\text{s}$ needed by ESP are most likely due to the fact that more measurements and outputs have been performed.

The Open MPI library again introduces the largest part of additional latency during message reception, namely after the data has been received from the socket. It should be investigated further, what causes this delay and whether it can be optimized.

Chapter 7: Summary and Conclusion

Within this theses an approach to reducing the message latency for Ethernet cluster communication has been followed. For that purpose the networking subsystem of the Linux kernel has been analyzed to find a solution most compatible with existing network devices. The solution found most suitable was to just circumvent the TCP/IP protocol stack and to retain both the standard network device interface and the socket interface for user-space applications. This allows for the use of standard network device drivers and provides applications with a well-known interface to the new communication means.

Since raw Ethernet cannot manage multiple communications over a single network interface due to its lack of a multiplexing facility, an additional, slim protocol suite atop Ethernet has been developed. At the first stage a basic datagram protocol (EDP) has been implemented. The benchmarks showed, that a significant latency reduction of about 30 %, compared to TCP, could be reached. A simple zero-copy approach was followed, but did not yield any further improvements.

As another result of the EDP benchmarks it was realized, that unreliable communication is not suitable for parallel processing, because even in closely interconnected networks data packets can be lost due to congestion or transmission errors. Therefore in a second step a streaming protocol (ESP) was developed providing sequenced and reliable data transfer. The it was tried to keep the implementation of ESP as slim as possible to retain the low latencies of the datagram protocol. Though this goal could be reached, the final benchmarks showed another issue. As ESP does not implement any flow control, it performs bad for large message sizes. The occurring retransmissions do largely influence the maximum achievable throughput. Therefore a future extension of the protocol could include a flow control algorithm optimized for the conditions of closely interconnected Ethernet networks to avoid the need of retransmissions as far as possible.

Both new protocols have been implemented into a single kernel module for version 2.6 Linux kernels. This allows for an easy installation into running systems, as there is no need for special hardware, kernel patches or additional drivers.

In the last step a communication module for the Open MPI message passing library has been ported atop the new streaming protocol. This could be achieved by applying a few changes to an existing TCP communication module, because ESP offers its services through network sockets, that behave similar to those for TCP. This also demonstrates the fact that the new protocols can easily be integrated into existing or new applications.

Appendix

A.1. References

- [1] Bonwick, J. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*.
<http://www.usenix.org/publications/library/proceedings/bos94/bonwick.html>
- [2] Jamal Hadi Salim, Robert Olsson, Alexey Kuznetsov. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference, Oakland, California, USA, November 2001*.
<http://www.usenix.org/publications/library/proceedings/als01/jamal.html>
- [3] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*, 2nd Edition. O'Reilly, 2002.
- [4] T. von Eicken, A. Basu, V. Buch, W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles, December 1995*.
<http://www.cs.cornell.edu/tve/u-net/papers/sosp.pdf>
- [5] Evolution of the Virtual Interface Architecture,
<http://www.cs.cornell.edu/vogels/papers/evolutionvia.pdf>
- [6] M-VIA, <http://crd.lbl.gov/FTG/MVIA/mvia.shtml>
- [7] MPICH: A Portable Implementation of MPI, <http://www-unix.mcs.anl.gov/mpi/mpich/>
- [8] MVICH: MPI for Virtual Interface Architecture,
<http://crd.lbl.gov/FTG/MVICH/mvich.shtml>
- [9] C. Csanady, P. Wycko.. Bobnet: High-Performance Message Passing for Commodity Networking Components. In *Proceedings of PDCN, December 1998*.
<http://www.osc.edu/~pw/bobnet/csanady-bobnet-pdcn98.pdf>
- [10] Myricom Home Page, <http://www.myri.com/>
- [11] MPLite: A Light-weight Message Passing Library,
http://www.scl.ameslab.gov/Projects/MP_Lite/
- [12] GAMMA: The Genoa Active Message MACHine,
<http://www.disi.unige.it/project/gamma/index.html>
- [13] G. Ciaccio. Optimal Communication Performance on Fast Ethernet with GAMMA. In *Proceedings of the Workshop PC-NOW, IPPS/SPDP'98, Orlando, Florida, April 1998*. <http://ipdps.cc.gatech.edu/1998/pc-now/ciaccio.pdf>
- [14] T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauer. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*.
- [15] The MPI/GAMMA Home Page,
<http://www.disi.unige.it/project/gamma/mpigamma/>

- [16] G. Chiola, G. Ciaccio. Porting MPICH ADI on GAMMA with Flow Contr. In *Proceedings of MWPP'99 (1999 Midwest Workshop on Parallel Processing), Kent, Ohio, August 11 - 13, 1999*.
<ftp://ftp.disi.unige.it/pub/project/GAMMA/mwpp99.ps.gz>
- [17] Piyush Shivam, Pete Wyckoff, D. K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of SC '01, Denver, CO, November 2001*. <http://www.osc.edu/~pw/emp/shivam-emp-sc01.pdf>
- [18] Netgauge: A network profiling tool, <http://www.unixer.de/~htor/linux/netgauge/>
- [19] Open MPI: Open Source High Performance Computing, <http://www.open-mpi.org/>
- [20] RFC 768: User Datagram Protocol, <http://rfc.net/rfc768.txt>
- [21] T. Höfler, W. Rehm. A Meta Analysis of Gigabit Ethernet over Copper Solutions for Cluster-Networking. In *Computer Architecture Technical Report*.
<https://www.tu-chemnitz.de/informatik/HomePages/RA/publications/p2004/cat2004-1.pdf>
- [22] RFC 791: Internet Protocol, <http://rfc.net/rfc791.txt>
- [23] RFC 1700: Assigned Numbers, <http://rfc.net/rfc1700.txt>
- [24] Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, <http://standards.ieee.org/getieee802/802.3.html>
- [25] The Ethernet, Physical and Data Link Layer Specifications Version 2.0, 1982
- [26] RFC 793: Transmission Control Protocol, <http://rfc.net/rfc793.txt>
- [27] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004*. <http://www.open-mpi.org/papers/euro-pvmmmpi-2004-overview/euro-pvmmmpi-2004-overview.pdf>
- [28] MPI-2: Extensions to the Message Passing Interface, July 1997, Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, July 1997. <http://www.mpi-forum.org>
- [29] Pallas GmbH, Pallas MPI Benchmarks - PMB, Part MPI-1. Technical report, 2000

A.2. List of Figures

Figure 2.1.1: The Linux Networking Subsystem.....	2
Figure 2.2.1: The Protocol Layer Socket and Private Data Structures.....	4
Figure 2.2.2: Linux Socket Buffer Pointer Operations.....	12
Figure 2.3.1: DMA ring (exemplary).....	17
Figure 2.3.2: Data Transmission and Reception Through the Linux Networking Subsystem	20
Figure 4.1.1: The EDP Datagram Structure.....	32
Figure 4.3.1: EDP Message Latency on Cluster-1.....	37
Figure 4.3.2: EDP Message Throughput on Cluster-1.....	37
Figure 4.3.3: EDP Message Latency on Cluster-2.....	38
Figure 4.3.4: EDP Message Throughput on Cluster-2.....	38
Figure 4.4.1: EDP Zero-copy Message Latency on Cluster-1.....	40
Figure 4.4.2: EDP Zero-copy Message Throughput on Cluster-1.....	41
Figure 4.4.3: EDP Zero-copy Message Latency on Cluster-2.....	41
Figure 4.4.4: EDP Zero-copy Message Throughput on Cluster-2.....	42
Figure 5.1.1: The ESP Packet Structure.....	46
Figure 5.2.1: The ESP Socket States.....	47
Figure 5.3.1: ESP Message Latency on Cluster-1.....	55
Figure 5.3.2: ESP Message Throughput on Cluster-1.....	56
Figure 5.3.3: ESP Message Latency on Cluster-2.....	56
Figure 5.3.4: ESP Message Throughput on Cluster-2.....	57
Figure 5.3.5: ESP ACK Processing Influence on Message Latency and Throughput on Cluster-1.....	57
Figure 5.3.6: Socket Buffer Size Influence on the ESP Maximum Message Throughput on Cluster-2.....	58
Figure 5.3.7: Socket Buffer Size Influence on the ESP Average Message Throughput on Cluster-2.....	58
Figure 6.1.1: The Modular Component Architecture.....	59
Figure 6.4.1: Results of the PMB Ping-Pong Benchmark.....	63
Figure 6.4.2: Results of the PMB Allgather Benchmark.....	63
Figure 6.4.3: Results of the PMB Reduce Benchmark.....	63
Figure 6.4.4: Results of the PMB Broadcast Benchmark.....	63
Figure 6.4.5: Profiling of an Open MPI Communication using TCP on Cluster-2.....	64
Figure 6.4.6: Profiling of an Open MPI Communication using ESP.....	65

A.3. List of Abbreviations

ACK	Acknowledgement
BTL	Binary Transmission Layer
EDP	Ethernet Datagram Protocol
ESP	Ethernet Streaming Protocol
IP	Internet Protocol
MPI	Message Passing Interface
NIC	Network Interface Card
PML	Point-to-Point Management Layer
RTO	retransmission timeout
RTT	round trip time
SRTT	smoothed round trip time
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

A.4. The Cluster Test Systems

Cluster-1:

- CPU 2 x AMD Athlon(tm) MP 1600+
- Main Memory 512 MB
- Network Interface SysKonnnect SK-98xx V2.0 Gigabit Ethernet Adapter
- Operating System Linux 2.6.9-22.ELsmp

Cluster-2:

- CPU 4 x Intel(R) Xeon(TM) CPU 2.40GHz
- Main Memory 2 GB
- Network Interface Intel Corporation 82544GC Gigabit Ethernet Controller
- Operating System Linux 2.6.9-22.ELsmp

Statutory Declaration

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, no other sources or auxiliary tools except those stated, referenced and acknowledged have been used.

Chemnitz, February 28, 2006

Mirko Reinhardt