Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report
Distributed Systems Lab

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Bär Jeremia, Föllmi Claudio

Bulk Transfer over Network

by

Achermann Reto, Kaufmann Antoine

Supervised by

Prof. Timothy Roscoe, Dr. Kornilios Kourtis

February 14, 2014

inf | Informatik
Computer Science

Contents

List of Figures

List of Tables

Part I

Bulk Transfer Specification

Acknowledgements

This part of the report contains the general specification of the bulk transfer system shared by both projects - the network backend as well as the shared memory backend. The specifications of the bulk transfer infrastructure were defined as a collaboration between the contributors of this part.

Contributors

- Achermann Reto <acreto@student.ethz.ch>
- Bär Jeremia <baerj@student.ethz.ch>
- Föllmi Claudio <foellmic@student.ethz.ch>
- Kaufmann Antoine <antoinek@student.ethz.ch>

Table of Contents

List of Figures

Chapter 1

Introduction

With an increasing number of cores a single computer can already be viewed as a distributed system which needs a distributed operating systems running on it [4]. Having several instances of processes running on different cores, the need for communication to synchronize them increases. Therefore a fast inter process communication mechanism is the basis for good performance.

However, distributing work among many processes running on different cores or even different machines also involves providing them with larger and larger amounts of data. This results in a second requirement for good performance: an efficient bulk transfer mechanism.

The bulk transfer should work seamlessly between domains running on the same machine as well as between different nodes in a network: A data block should be movable between multiple domains without copying it and in addition to that sent over the network without copying it to transmit buffers.

A fast bulk transfer implementation is not useful, when the two participating domains do not trust each other and the implementation does not provide mechanisms to enforce certain policies: there must be a possibility to ensure data integrity once the data block has been sent.

Report Outline In this report, we will give an overview of related work 2 followed by a brief introduction to the Barrelfish related components in Chapter 3. In Chapter 4 we give a high level description of the architectural design followed by a formal specification of the semantics 5. We provide an overview of our sample implementation in Chapter 6 followed by the description of a sample application which uses our bulk transfer implementation in Chapter 7. The last chapter briefly discusses our conclusions of our approach.

Chapter 2

Related Work

fbufs The article *Fbufs: A High-Bandwidth Cross-Domain Transfer Facility* [9] presents an I/O buffer management facility providing high throughput for I/O intensive applications. Its focus is on *fast buffers* (fbufs) crossing multiple domains and being processed on the way, as is the case for data traversing from an application to the network and vice versa. I/O data in fbufs is accessible read-only as soon as buffers are shared with other domains. Their size is a multiple of the system's page size. This allows to use shared memory and page remapping technology to avoid the actual copying of data. The read-only property ensures data integrity. To allow data modification on-the-fly, references to sections of immutable buffers are combined into an aggregate object tree. This allows for dynamic recombination of data chunks making the data changeable on a higher level of abstraction. The article makes a strong point that increased sharing of buffers will benefit speed but at the same time start to compromise security.

Many of the ideas in the paper were integrated into our design of the bulk transfer interface, especially with focus on allowing eager optimization for the shared memory scenario.

rbufs The design of rbufs [6] consists of two different types of memory regions. First there is a data area, which is a large contiguous range of virtual memory. The protection is the same for the entire data area such that the data source can write and the other can at least read. In general, the data area can always be written by the generating domain and therefore is volatile.

Secondly, there are control areas at each side of the channel which can be viewed as a circular buffer. Two of these control areas form a channel. The protection of these control areas depend on the direction of data transfer: The control areas must be at least writable by the writing domain and readable by the receiving domain.

Data transfers are represented by iorecs which are references to a number of regions in the data area, allowing for the aggregation of multiple regions in the data area.

Rbufs can be used to form longer channels by spanning them over multiple domains.

mbufs In FreeBSD, the kernel uses mbufs [19] as a basis for IPC and network processing: the arrived packets are stored in (potentially) multiple mbufs which

are chained together. As with pbufs, the chaining allows to efficiently remove headers and footers from the received buffers. The fact that the mbufs are used by the kernel only leads to a need for copying the data to user space.

Xen Grant Tables In XEN [2] - a bare metal hypervisor - domains[1] can share frames with other domains by explicitly granting access to them. There is a grant table associated with each domain specifying which frames can be accessed by this domain. Granting access to another domain means allowing access to the granter's memory. This can be viewed as a capability.

The access rights can be passed to another domain by invoking XEN to set the corresponding entry in the grant table i.e. transferring the capability to access that frame to another domain. That way a shared frame can be set up which can be used to transfer data between domains.

This shared frame allows virtual machines residing on the same physical machine to share data with each other.

[1] Domain in this case means a virtual machine running on top of XEN

Chapter 3

Barrelfish Operating System

The Barrelfish OS[1] is a research operating system developed as a collaboration between the Systems Group[2] at ETH Zurich and Microsoft Research[3]. In this Section, we will briefly describe important aspects of Barrelfish with respect to a bulk transfer infrastructure.

3.1 Operating System Structure

Most of today's popular operating systems such as Linux, Microsoft Windows or Mac OS X have a monolithic kernel architecture. In contrast to that, Barrelfish has a multikernel approach [21]. The multikernel architecture can be viewed as a collection of multiple microkernels - one per core - which communicate via explicit messages to keep the OS state consistent, rather than accessing shared data structures.

A microkernel provides only a very small set of security relevant services to the applications, such as setting up page tables or enabling inter process communication. All the other services like memory management or device drivers run entirely in user space. The multikernel approach increases the need for communication between domains. This demands for an efficient way of communication, and not only for small control messages but also for bulk data.

3.2 Flounder

In Barrelfish, inter process communication is done by exporting a certain interface which can be invoked by other domains. The interfaces are described in a domain specific language and compiled into C code using a tool called flounder [3]. As soon as a domain exported the interface, another domain can bind

[1] `http://www.barrelfish.org`

[2] Systems Group, Department of Computer Science, CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland

[3] `http://research.microsoft.com/`

to it and send messages, either in an asynchronous fashion or with full RPC semantics.

3.3 Resource Management and Capabilities

In Barrelfish, critical system resources such as physical memory or kernel services are protected using capabilities [1]. A capability grants the possessing domain certain access rights to a resource e.g. reading a certain memory range. This access rights can be granted to other domains by passing the capability to the other domain.

Accessing system resources such as kernel services are done by capability invocations instead of traditional systems calls. That way only if the domain possesses the capability can it do a system call.

Capabilities are strongly typed and have certain access rights associated with them. For instance, a RAM capability can be retyped into a frame capability by doing an invocation on it. The access rights of a frame capability can then be restricted to read only. These operations are usually one way.

3.4 Current Bulk Transfer Implementation

The technical note about bulk transfer [22] describes the two existing bulk transfer mechanisms in Barrelfish and their limitations. We want to briefly describe them here and explain their drawbacks.

3.4.1 The Official Bulk Transfer Infrastructure

This implementation of a bulk transfer mechanism is based on a single shared capability which is mapped read-write in both domains. The mapped virtual address range has to be contiguous. The mapped memory range is divided into blocks of equal size. To do a data transfer, data is written into one of the blocks and its block id transmitted to the other domain.

Limitations

There are several limitations with the current state of this implementation. We want to briefly list them here.

- **Security:** cannot be enforced. Both domains have to trust each other not to mess with the blocks.
- **Multiple Domains:** It is possible to use the shared capability with more than two domains. However, all of them must be trusted to follow the protocol and track the location of the buffers by themselves.
- **Copy Semantics:** There is no copy operation that guarantees the copying domain that the data will be kept consistent.
- **Abstraction:** The interface is modelled after the underlying implementation and relies on shared memory. This is in stark contrast to other means of communication in Barrelfish, like flounder message passing.

3.4.2 Networking Bulk Transfer: pbufs

The concepts of pbufs is similar to the official implementation described above: there is a shared frame used as buffer. However, in contrast the memory locations do not need to be consecutive. Furthermore, there is another layer of indirection that enables the replacement of one memory location by another to reuse the pbuf meta structure.

Limitations

As with the official implementation, there are some limitations with the pbufs as well. We will quickly summarize them here.

- **Security:** Shared memory region is mapped read/write in both domains.
- **Multiple Domains:** It should work in theory, but it is hard to tell when the memory can be reused. Furthermore, all domains must be co-operative
- **Memory Reclamation:** It's easy to get it wrong, leading to memory leaks because it's hard to tell when all threads are done with accessing the buffer.

3.4.3 Summary

To sum up, the two available bulk transfer implementations in Barrelfish require the participating domains to co-operate and follow the protocol. The lack of policy enforcement makes it not possible to use the implementations in a malicious environment. In addition, the complexity of having multiple domains sharing buffers increases and it is hard to tell which buffers are allowed to be reused.

Chapter 4

Architectural Design

This Chapter describes the major building blocks of our bulk transfer infrastructure. We will give an explanation of the used terms and state the goals of the bulk transfer implementation.

4.1 Design Goals

The technical note 014 [22] about bulk transfer states some of the goals for a bulk transfer implementation, which we have adapted and complemented:

1. Avoid data copy as much as possible, if it can't be avoided, then try to push it into user-space/user-core.
2. Ability to batch notifications.
3. Should work with more than two domains.
4. Should work with multiple producers and multiple consumers.
5. True zero copy capability (scatter-gather packet sending/receiving).
6. Should support different means of data transfer e.g. shared memory or network.
7. Unified interface for all possible backend implementations.

4.2 Terminology

4.2.1 Backend

The bulk transfer backend is the implementation or hardware specific part of the bulk transfer infrastructure. It handles the operations and events triggered by the user and forwards them to the destination. To list some possible backends:

- Shared memory between two domains on the same machine.
- Network between two machines reachable over the network.
- Local (between threads in a single domain)

- Direct Memory Access Engines
- Mailboxes between two different cores (e.g. OMAP44xx Cortex A9 and Cortex M3.)
- GPU / Accelerators

4.2.2 Endpoint

A `bulk_endpoint` represents either the source or destination of a data flow. An endpoint can either be local i.e. created by the domain or remote, created by another domain. The local endpoint may be created implicitly depending on the remote endpoint. Endpoints are backend specific and therefore specify how the data is to be transferred or received. An endpoint is either the source or sink of a data transfer.

Endpoint Descriptors

Bulk endpoints are referred to by endpoint descriptors which contain the necessary information to enable the channel operations.

4.2.3 Channel

A `bulk_channel` consists of exactly two endpoints which must exists before a channel can be created. A channel is just a point-to-point link and spans at most two domains[1]. Each channel has a clearly specified direction of data transfer which is either receive or transmit, and the endpoints must be the corresponding source or sink. An illustration of the channel, endpoint and library relationship can be seen in Figure 4.1.

The channel initialization consists of two different operations: create and bind.

Channel Creation

The channel create operation is the first to be executed to establish a new bulk channel. The application needs to specify the local endpoint for this channel which has to be created beforehand. The endpoint type also determines the channel type. The general channel parameters are set and chosen during the create procedure.

Channel Binding

The second step in the channel initialization protocol is the binding process. In contrast to the create procedure, the application needs to provide an endpoint descriptor of the remote endpoint. Local endpoints are created implicitly depending on the remote endpoint. The binding side of a channel has to adapt the channel properties as defined by the creation side.

[1] A local channel resides in just one domain

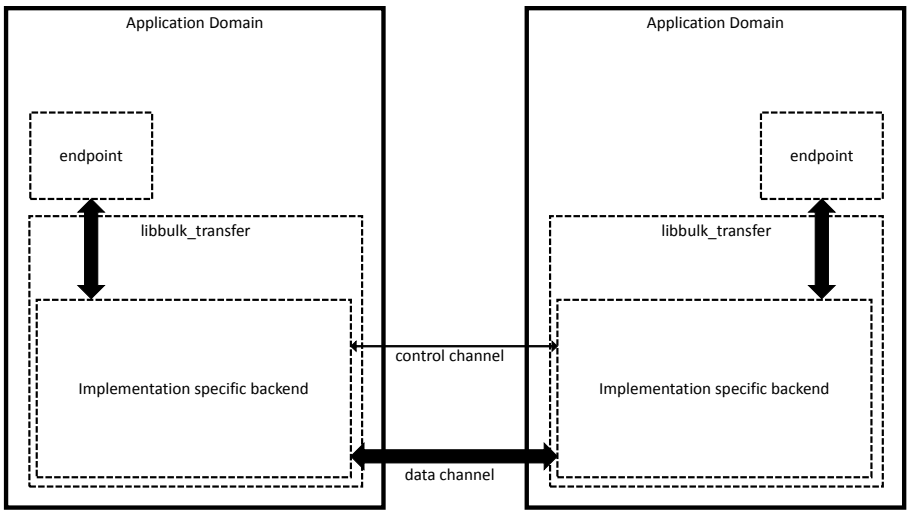

Figure 4.1: Bulk Transfer Architecture

Data and Control Channel

On a conceptual level, each channel consists of a control channel for small messages and a data channel used for the actual data transfer. Depending on the implementation, the control channel may be in-line with the data channel or the data channel might just be virtual.

4.2.4 Channel Properties

Each channel has certain properties by which both endpoints of the channel must comply.

Roles

After the channel is established, each endpoint is either the `MASTER` or `SLAVE` with respect to that channel. During the channel creation the local endpoint may be in the `GENERIC` role, which enables the binding side to decide the role distribution.

Trust Levels

The channel creator decides on the trust level of the channel. The trust level specifies the level of data integrity protection applied to this channel. There are three different trust levels:

1. **Full Trust:** There are no protection mechanisms[2] enforced. The domains are trusted not to mess around with the data blocks after they are transferred to another domain.

[2]Depending on the backend, such protection mechanisms could be removing the MMU mapping or restricting the memory access for hardware devices (IO-MMU)

2. **Half Trust:** The basic protection mechanisms are applied but the capabilities are not revoked. The application may regain access to the data blocks manually. This should only be seen as a protection against accidental overwriting of buffers.

3. **No Trust:** In addition to the application of the basic protection mechanisms, the change in the access rights to the resources are enforced i.e. the corresponding capabilities are revoked.

4.2.5 Pool

A pool represents a contiguous range of virtual memory and consists of a number of equal sized parts called `bulk_buffer` (refer to Section 4.2.6). The number and size of the buffers are specified upon pool allocation. In order to use the pool over a channel, it needs to be assigned first. After the pool is assigned to a channel, the pool occupies a contiguous range of virtual memory in both domains.

Pool ID

To uniquely identify the pool across domains, cores and even machines, we assign each pool a unique id. The pool identifier is generated at allocation time.

Pool Trust Level

As with the channels, the pools also have a trust level. The trust level of a pool is set to the channel trust level upon the first assignment. In general, the trust level of the pool must match the trust level of the channel.

4.2.6 Buffer

A `bulk_buffer` is the smallest unit of transfer in our infrastructure. Each buffer is a contiguous region of virtual and physical memory. As mentioned in the previous section, the size of the buffers can be specified upon pool allocation. In order to guarantee the enforcement of access rights we require the buffer size to be a multiple of page size[3]. A buffer can either be present in the domain i.e. its data is accessible or it is not present in the domain and the data is not accessible.

Ownership

The ownership of a buffer specifies which operations a domain can do with the buffer. There exists exactly one owner for each buffer at any point of time.

4.3 Bulk Transfer Architecture

We suggest to have a layered architecture as shown in Figure 4.2. With a concrete application in mind there are 3+1 layers:

[3]With the capability system the size must also be a power of two.

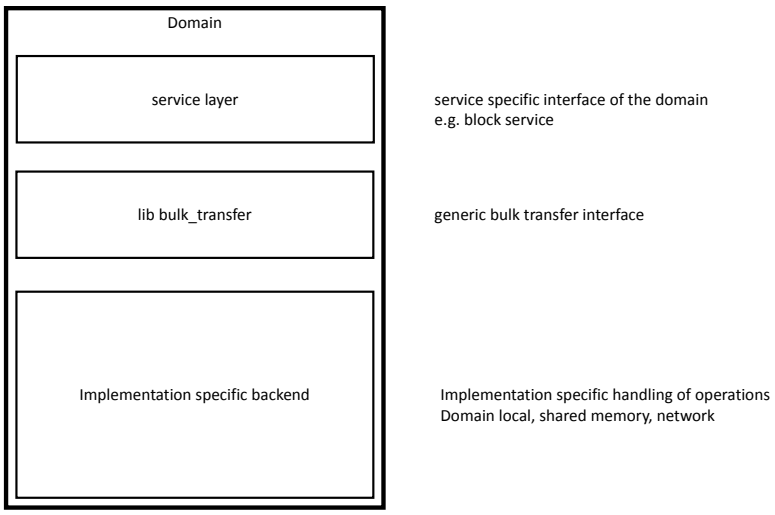

Figure 4.2: Architecture Layers

1. **Application Layer**[4]**:** The user application which makes use of the bulk transfer library and the service
2. **Service Layer:** The interface of a service that is used by the application e.g. a block service.
3. **Bulk Transfer Library:** Provides the unified interface to the backend, common functions and does general checks.
4. **Implementation Specific Backend:** This layer deals with the effective data transfer between the endpoits.

4.4 Additional Libraries

The core bulk transfer library just provides the functionality necessary to create channels and transfer buffers. We want to give the application freedom how it allocates and manages pools or send more data than a buffer can hold. There are two suggestions we present here that complement the bulk transfer library.

Pool Allocator

This library is responsible for allocating pools and their resources. We provide a sample implementation of a pool allocator. The reason why we decided that the pool allocator is not part of the core library is that an application may want to have another way of managing the buffers of the pool or use a lazy approach in resource allocation which may be more suitable for the usage scenario.

Our allocator will just allocate all the resources for the pool, map the buffers and fills the list of free buffers.

[4]This is the +1

Aggregates

The core interface of the bulk transfer architecture does only support the sending or receiving of one buffer at a time. To overcome this limitation, we suggest a library which works on top of that that allows sending larger amounts of data by the use of buffer aggregates which may be similar to the aggregate objects used in [9]. Note that this has not been implemented yet.

Aggregates should make it possible to send a collection of bulk buffers over a channel while the buffers may belong to different pools. One approach to this would be to send an additional data structure before or after the buffers are being transmitted. This data structure may look like the one in Listing 4.1. Please refer to Section 8.4 for a further discussion.

```
1 struct bulk_aggregate_object
  {
3     size_t num_bufs;
      struct
5     {
          struct bulk_pool_id pool_id;
7         size_t              buffer_id;
      }       buffers[];
9 }
```

Listing 4.1: Possible Representation of an Aggregate Object

Chapter 5

Semantic Specification

One of the goals of a bulk transfer mechanism is to provide the possibility to enforce certain policies such as restricting access rights. This requires a clear understanding at what point which domain has access to which resources. In this Chapter we present a formal specification of our model (Section 5.1) as well as an overview of the channel operations (Section 5.2)

Notation	Meaning
$\mathrm{dom}\, r$	Domain of relation r
$r[x]$	Relational image of values x in r
$r \circ s$	Composition of relations r,s
r^{-1}	Inverse of relation r
$A \to B$	Set of functions from A to B
$A \nrightarrow B$	Set of partial functions from A to B

Table 5.1: The used formal specification

5.1 Formal Specification

In this Section we want to give a high level description of the possible transfer types together with a precise formal specification for each of the transfer types. We aimed to keep this formal specification as open as possible to make it suitable for any possible backend implementation. Note that the specification specifies the system state and not the state as viewed from a particular domain. The notation used can be seen in Table 5.1.

5.1.1 Buffer States

For every domain, each buffer is always in its clear specified state.[1] These states are:

1. **Invalid:** the buffer is not present in this domain

[1] Note that this state may be different in different domains.

2. **Read/Write:** the buffer is owned by this domain and accessible for reading and writing
3. **Read Only Owned**: the buffer is owned by this domain but only accessible for reading[2]
4. **Read Only** : the buffer is accessible for reading

5.1.2 Formal State Definition

In order to model the operations we must have a clear understanding of the model state in our formal specification. The state of the model is expressed in Algorithm 1. To make it clear, we would like to highlight some of the statements:

- **Buffer Copies:** (line 14) Conceptually, there exists a graph of channels. This function tracks over which channels this buffer has been copied.
- **Queues:** (line 19 and 20) These queues represent the data channel and the control channel.
- **Endpoints:** (line 17 and 18) These partial bijections return the endpoint given a channel and a direction or role.

Buffer States

For every domain, each buffer is always in its clear specified state.[3] These states are:

1. **Invalid:** the buffer is not present in this domain
2. **Read/Write:** the buffer is owned by this domain and accessible for reading and writing
3. **Read Only Owned**: the buffer is owned by this domain but only accessible for reading[4]
4. **Read Only** : the buffer is accessible for reading

5.1.3 Invariants

From the state definition we can formulate invariants which hold at any time. These invariants are listed in Algorithm 2. These invariants are important for a proper understanding of the bulk transfer model and we give a short explanation for the invariants here.

1. Specifies all the domains that hold copies of a buffer b. We take all the channels, which received a buffer copies and take only the sink endpoint and do a domain lookup.

[2]This state is added for the clarification of an owned copy
[3]Notice that this state may be different in different domains.
[4]This state is added for the clarification of an owned copy

Algorithm 1 State Definition

1: $\text{ACCESS\_RIGHTS} := \{\text{NONE}, \text{READ}, \text{READ\_WRITE}\}$
2: $\text{ROLES} := \{\text{MASTER}, \text{SLAVE}\}$
3: $\text{DIRECTIONS} := \{\text{SOURCE}, \text{SINK}\}$
4:
5: $\text{valid\_buffers} \subseteq \text{BUFFERS}$
6: $\text{valid\_pools} \subseteq \text{POOLS}$
7: $\text{valid\_channels} \subseteq \text{CHANNELS}$
8: $\text{valid\_eps} \subseteq \text{ENDPOINTS}$
9:
10: $\text{buffers\_to\_pools} \in \text{valid\_buffers} \to \text{valid\_pools}$
11: $\text{buffer\_ownership} \in \text{valid\_buffers} \to \text{DOMAINS}$
12: $\text{buffer\_access} \in (\text{valid\_buffers} \times \text{DOMAINS}) \to \text{ACCESS\_RIGHTS}$
13: $\text{buffer\_data} \in \text{valid\_buffers} \to \text{DATA}$
14: $\text{buffer\_copies} \subseteq \text{valid\_buffers} \times \text{valid\_channels}$
15: $\text{pools\_to\_channels} \subseteq \text{valid\_pools} \times \text{valid\_channels}$
16: $\text{ep\_to\_dom} \in \text{valid\_ep} \to \text{DOMAINS}$
17: $\text{dir\_to\_ep} \in (\text{valid\_channels} \times \text{DIRECTIONS}) \nrightarrow \text{valid\_ep}$
18: $\text{roles\_to\_ep} \in (\text{valid\_channels} \times \text{ROLES}) \nrightarrow \text{valid\_ep}$
19: $\text{data\_queues} \in \text{valid\_channels} \to \text{BUFFER\_QUEUES}$
20: $\text{control\_queues} \in \text{valid\_channels} \to \text{BUFFER\_QUEUES}$

Algorithm 2 Invariants

1: $\text{copy\_receivers}(b, \text{buffer\_copies}') :=$
$\text{ep\_to\_dom}[\text{dir\_to\_ep}[\text{buffer\_copies}'[\{b\}] \times \{\text{SINK}\}]]$
2: $\forall\, b, d : \text{buffer\_ownership}(b) = d \wedge b \notin \text{dom}(\text{buffer\_copies}) \Leftrightarrow$
$\text{buffer\_access}(b, d) = \text{READ\_WRITE}$
3: $\forall\, b, d : (\text{buffer\_ownership}(b) = d \wedge b \in \text{dom}(\text{buffer\_copies})) \vee$
$d \in \text{copy\_receivers}(b, \text{buffer\_copies}) \Leftrightarrow \text{buffer\_access}(b, d) = \text{READ}$
4: $\forall\, b, d : \text{buffer\_ownership}(b) \neq d \wedge d \notin \text{copy\_receivers}(b, \text{buffer\_copies}) \Leftrightarrow$
$\text{buffer\_access}(b, d) = \text{NONE}$
5: $\text{copy\_reachability\_satisfied}(\text{buffer\_copies}') :=$
$\forall\, b, c : (b, c) \in \text{buffer\_copies}' \Rightarrow$
$\exists\, d : d = \text{ep\_to\_dom}(\text{dir\_to\_ep}(c, \text{SOURCE})) \wedge$
$(\text{buffer\_ownership}(b) = d \vee d \in \text{copy\_receivers}(b, \text{buffer\_copies}'))$

2. This invariant says, that if a domain has buffer ownership and if there are no copies of this buffer, then this domain has read-write access to the buffer.

3. A domain has read access to a buffer, if this domain owns the buffer and there are copies around. A domain also has read access if it receives a copy.

4. A domain has no access to a buffer, if this domain has never received a copy and it is not the owner of the buffer

5. This invariant says, that if there is a buffer copy, there exists a directed path from the owner along the graph edges to this copy i.e. a copy is always reachable by its owner.

5.1.4 Channel Creation and Binding

A channel is established by a create (Algorithm 3) and a bind operation (Algorithm 4). The creation of a channel requires that the endpoint associated with the channel has not yet been used for a channel yet. After the creation, the channel is valid, the second endpoint has not been assigned and the other role exists not for this endpoint.

The binding requires that there exists a valid channel with the specified remote endpoint and that the local endpoint has not been assigned to a channel yet. Further, the direction and roles of this endpoint must not have been present in the state. After the binding, the roles and directions are assigned and the remote endpoint is not modified.

Algorithm 3 Create Channel

Require: domain $\in$ DOMAINS
Require: ep $\in$ (ENDPOINTS $\setminus$ valid_eps)
Require: direction $\in$ DIRECTIONS
Require: role $\in$ ROLES
1: **procedure** ChannelCreate(domain, ep, direction, role)
2: **end procedure**
Ensure: ep $\in$ valid_eps$'$
Ensure: $\exists$ c : c $\notin$ valid_channels $\wedge$ c $\in$ valid_channels$'$
Ensure: dir_to_eps$'$(c, direction) = ep
Ensure: roles_to_eps$'$(c, role) = ep
Ensure: (c, other_dir(direction)) $\notin$ dom(dir_to_eps$'$)
Ensure: (c, other_role(role)) $\notin$ dom(roles_to_eps$'$)
Ensure: eps_to_domains$'$(ep) = domain

5.1.5 Pool Allocation and Assignment

Sending data over channels requires the allocation and assignment of pools to the channels. The allocation of a pool (Algorithm 5) requires that the pool to create is not valid and there is at least one unused (not valid) buffer to be associated with this pool. After the creation, all buffers of this pool are owned by this domain and belong to the newly created pool.

Algorithm 4 Bind to Channel

Require: domain ∈ DOMAINS
Require: local_ep ∈ (ENDPOINTS \ valid_eps)
Require: remote_ep ∈ valid_eps
Require: ∃ chan, remote_dir, remote_role :
Require: dir_to_eps(chan, remote_dir) = remote_ep
Require: roles_to_eps(chan, remote_role) = remote_ep
Require: (chan, other_dir(remote_dir)) ∉ dom(dir_to_eps)
Require: (chan, other_role(remote_role)) ∉ dom(roles_to_eps)
1: **procedure** ChannelBind(domain, local_ep, remote_ep)
2: **end procedure**
Ensure: local_ep ∈ valid_eps'
Ensure: dir_to_eps'(chan, other_dir(remote_dir)) = local_ep
Ensure: dir_to_eps'(chan, remote_dir) = remote_ep
Ensure: roles_to_eps'(chan, other_role(remote_role)) = local_ep
Ensure: roles_to_eps'(chan, remote_role) = remote_ep
Ensure: eps_to_domains′(local_ep) = domain

The assignment of a pool to a channel (Algorithm 6 requires that the pool has not been assigned to this channel yet and that the channel is bound.

Algorithm 5 Create Pool

Require: domain ∈ DOMAINS
Require: pool ∈ (POOLS \ valid_pools)
Require: bufs ⊆ (BUFFERS \ valid_buffers) ∧ bufs ≠ ∅
1: **procedure** PoolCreate(domain, pool, bufs)
2: **end procedure**
Ensure: pool ∈ valid_pools′
Ensure: bufs ⊆ valid_buffers′
Ensure: ∀ b : b ∈ bufs ⇒ buffer_to_pools(b) = pool
Ensure: ∀ b : b ∈ bufs ⇒ buffer_ownership(b) = domain

5.1.6 Buffer Transfer Types

Buffer Transfers are the core functionality of a bulk transfer mechanism: sending data from one domain to another. There exist different possibilities which we describe in the following Sections. Figure 5.1 shows the relationship between the buffer states and the transfer operations.

5.1.7 Moving a Buffer on a Channel

Algorithm 7 describes the semantics of a buffer move operation. A buffer can only be movable if the channel is bound and the pool this buffer belongs to is assigned to the channel. In addition to that, the domain must be the owner of the buffer and the direction of the channel must be transmit i.e. the local endpoint is the source.

Algorithm 6 Assign Pool to Channel

Require: ep $\in$ valid_eps
Require: pool $\in$ valid_pools
Require: $\exists\, \text{chan}, \text{dir} : \text{dir\_to\_eps}^{-1}(\text{ep}) = (\text{chan}, \text{dir})$
Require: $(\text{pool}, \text{chan}) \notin \text{pools\_to\_channels}$
1: **procedure** ChannelAssignPool(ep, pool)
2: **end procedure**
Ensure: $\text{pools\_to\_channels}^{-1}[\{\text{chan}\}] = \text{pools\_to\_channels}^{-1}[\{\text{chan}\}] \bigcup \{pool\}$

After the move is done, the buffer is sent over the data channel and its contents are preserved. The ownership of the buffer is transferred to the receiving domain unless the contents need to be copied in a new buffer.

Algorithm 7 Move Buffer on Channel

Require: ep $\in$ valid_ep
Require: $\exists\, \text{domain} : \text{ep\_to\_dom}(\text{ep}) = \text{domain}$
Require: $\text{buffer\_ownership}(\text{buffer}) = \text{domain}$
Require: $\exists\, \text{chan} : \text{dir\_to\_ep}^{-1}(\text{ep}) = (\text{chan}, \text{SOURCE})$
Require: $\exists\, \text{other\_ep} : \text{dir\_to\_ep}(\text{chan}, \text{SINK}) = \text{other\_ep}$
Require: $(\text{buffer}, \text{chan}) \in \text{buffers\_to\_pools} \circ \text{pools\_to\_channels}$
1: **procedure** ChannelMoveBuffer(ep, buffer)
2: **end procedure**
Ensure: $\exists\, \text{buf} : \text{data\_queues}'(\text{chan}) = \text{enqueue}(\text{data\_queues}(\text{chan}), \text{buf})$
Ensure: $\text{buffer\_data}'(\text{buf}) = \text{buffer\_data}(\text{buffer})$
Ensure: $\text{buffer\_ownership}'(\text{buf}) = \text{ep\_to\_dom}(\text{other\_ep})$
Ensure: $\text{buf} \neq \text{buffer} \Rightarrow \text{buffer\_ownership}'(\text{buffer}) = \text{ep\_to\_dom}(\text{ep})$

5.1.8 Buffer Pass

A domain has the possibility to pass the ownership of a buffer to another domain i.e. enabling read-write access. This operation (Algorithm 8 is conceptually like a move with the difference that the local endpoint must be the sink of the channel.

After the pass is executed, the buffer is enqueued to the control queue and the buffer ownership is transferred to the receiving domain.

5.1.9 Read-Only Copy

The basic requirements of the copy operation (Algorithm 9 are almost the same as with the move operation, with one fundamental difference: The buffer to be copied is either owned by the domain, or the domain received a copy of the buffer.

In contrast to a move, the ownership of the buffer does not change when doing a copy. Further the copy is tracked. It may be the case that the data needs to be copied into another buffer. Then the ownership is also passed.

Algorithm 8 Pass Buffer on Channel

Require: $ep \in valid\_ep$
Require: $\exists\, domain : ep\_to\_dom(ep) = domain$
Require: $buffer\_ownership(buffer) = domain$
Require: $\exists\, chan : dir\_to\_ep^{-1}(ep) = (chan, SINK)$
Require: $\exists\, other\_ep : dir\_to\_ep(chan, SOURCE)) = other\_ep$
Require: $(buffer, chan) \in buffers\_to\_pools \circ pools\_to\_channels$
1: **procedure** ChannelPassBuffer(ep, buffer)
2: **end procedure**
Ensure: $control\_queues'(chan) = enqueue(control\_queues(chan), buffer)$
Ensure: $buffer\_ownership'(buffer) = ep\_to\_dom(other\_ep)$

Algorithm 9 Copy Buffer on Channel

Require: $ep \in valid\_ep$
Require: $\exists\, domain : ep\_to\_dom(ep) = domain$
Require: $buffer\_ownership(buffer) = domain \vee$
$(\exists\, c : (buffer, c) \in buffer\_copies \wedge$
$ep\_to\_dom(dir\_to\_ep(c, SINK)) =$
$ep\_to\_dom(ep))$
Require: $\exists\, chan : dir\_to\_ep^{-1}(ep) = (chan, SOURCE)$
Require: $\exists\, other\_ep : dir\_to\_ep(chan, SINK) = other\_ep$
Require: $(buffer, chan) \in buffers\_to\_pools \circ pools\_to\_channels$
1: **procedure** ChannelCopyBuffer(ep, buffer)
2: **end procedure**
Ensure: $\exists\, buf : data\_queues'(chan) = enqueue(data\_queues(chan), buf)$
Ensure: $buffer\_data'(buf) = buffer\_data(buffer)$
Ensure: $buffer\_ownership'(buffer) = buffer\_ownership(buffer)$
Ensure: $buf \neq buffer \Rightarrow buffer\_ownership'(buf) = ep\_to\_dom(other\_ep)$
Ensure: $buf = buffer \Rightarrow$
$buffer\_copies'[\{buffer\}] = buffer\_copies[\{buffer\}] \cup \{chan\}$

5.1.10 Copy Release

Doing a release (Algorithm 0) of a copy is only allowed if there are no other copies originating from this domain, i.e. the release does not violate the reachability constraints. Further the buffer needs to be copied to the domain (not moved). When the owner of the copy gets all its copies back by a release, the buffer is changed to read-write again and its copy status is removed.

Algorithm 10 Release Copied Buffer on Channel

Require: $\text{ep} \in \text{valid\_ep}$
Require: $\exists\, \text{chan} : \text{dir\_to\_ep}^{-1}(\text{ep}) = (\text{chan}, \text{SINK})$
Require: $\exists\, \text{other\_ep} : \text{dir\_to\_ep}(\text{chan}, \text{SOURCE})) = \text{other\_ep}$
Require: $(\text{buffer}, \text{chan}) \in \text{buffer\_copies}$
Require: $\text{copy\_reachability\_satisfied}(\text{buffer\_copies} \setminus \{(\text{buffer}, \text{chan})\})$
Require: $(\text{buffer}, \text{chan}) \in \text{buffers\_to\_pools} \circ \text{pools\_to\_channels}$
1: **procedure** ChannelReleaseCopy(ep, buffer)
2: **end procedure**
Ensure: $(\text{buffer}, \text{chan}) \notin \text{buffer\_copies}'$
Ensure: buffer_access(buffer, domain) = NONE

5.1.11 Full Copy

When doing a full copy, the buffer will be accessible read-write in both domains in the end while ensuring data integrity at the sender. This can be conceptually viewed as a local copy[5] combined with a move. Therefore with a full copy a real memory-to-memory copy cannot be avoided and the receiving domain will get a buffer moved event.

5.2 Overview of Operations

The previous section gave a precise specification of the channel operations and their semantics. To clarify and to show the intended flow of steps we illustrate certain operations in this Section.

5.2.1 Buffer State Diagram

The formal specification clearly defines the possible states of a buffer in the whole system. From a domain point of view, the buffer states and their transitions can be seen in Figure 5.1. Notice, that we have introduced another state `READ_ONLY_OWNED` to distinguish the domain that first copied a buffer.

The state diagram also summarizes all possible operations and events that can occur on a channel.

5.2.2 Channel Setup

Establishing a new bulk channel between two endpoints involves two operations to be executed, a `channel_create` followed by a `channel_bind`. The workflow of a channel setup can be seen in Figure 5.2.

[5] e.g. memcopy

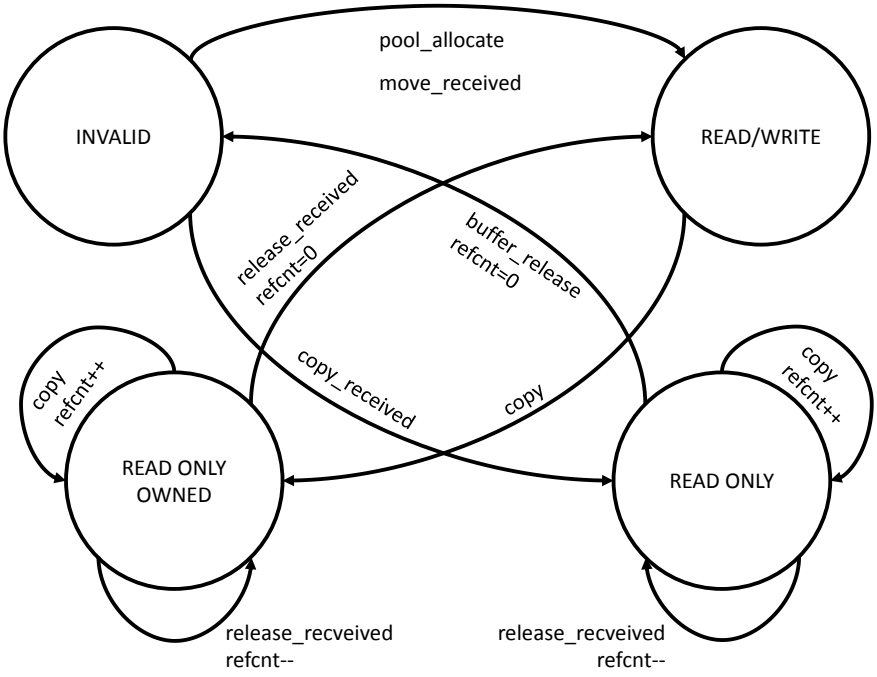

Figure 5.1: Buffer State Diagram (Domain View)

Creator Side

1. **Endpoint Create:** Initialization of a new local endpoint
2. **Exporting Endpoint:** Export of the information about the local endpoint to a name service.
3. **Channel Create:** Invocation of `bulk_channel_create(...)` with the callbacks and the setup parameters as arguments
4. **Backend:** After general checks and setup, the call gets forwarded to the implementation specific backend which enters a listening mode
5. **Bind Request:** The other side has sent a bind request. This request is forwarded to the application by invoking the callback handler and a reply is sent back.

Binding Side

1. **Remote Endpoint Creation:** To bind a channel, a remote endpoint is needed. There are two ways to do this:
   (a) Look up: Querying the name service for the endpoint information with the name of the exported service.
   (b) Explicitly created with pre-defined values.

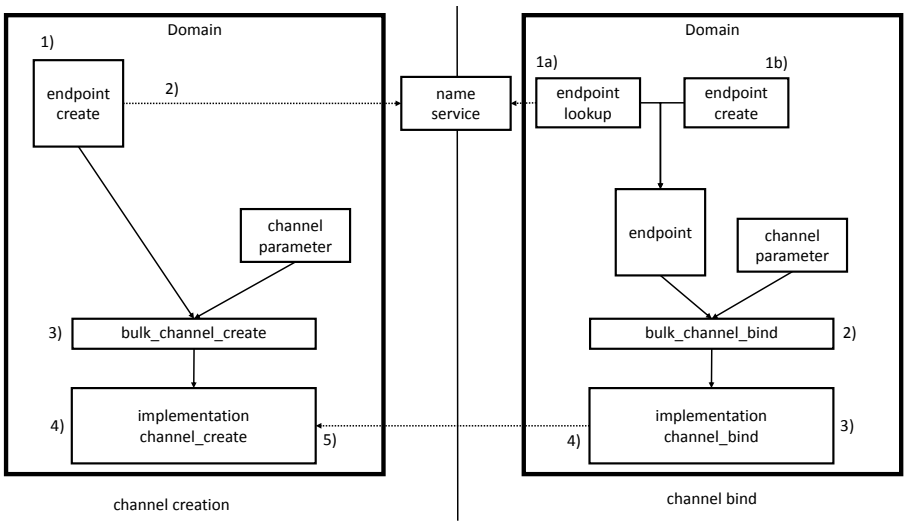

Figure 5.2: Setup of a Bulk Channel

2. **Channel Bind:** Invocation of `bulk_channel_bind(...)` with the callbacks and binding parameters as arguments

3. **Backend:** After general checks and binding steps, the backend specific implementation code is invoked.

4. **Send Bind Request:** The binding message is sent to the remote endpoint

5. **Bind Reply:** The bind reply is received and the continuation is executed.

5.2.3 Pool Assignment

The pool assignment procedure can be seen in Figure 5.3. In general, the channel master needs to provide buffers for operating the channel and thus is expected to add pools[6]. The pool assignment does not change anything in the ownership of the buffers.

Note that the pool assignment procedure is a two phase protocol, where the other side can veto the assignment request. The steps, as shown in Figure 5.3 are:

1. **Application:** Either the pool is allocated by the assigning domain or the pool was received earlier by another domain.

2. **Bulk Transfer Library** By invoking `bulk_channel_pool_assign(..)` the assignment process starts and general checks are done within the library.

[6]There is no restriction that the slave endpoint can't add pools as well.

3. **Backend:** After the checks, the backend specific pool assignment handler is invoked, which sends a pool assignment request to the other endpoint
4. **Backend:** The other side receives the pool assignment request
5. **Bulk Transfer Library:** The pool resources are allocated by the bulk transfer library[7] and the pool added to the domain list.
6. **Callback:** if everything succeeded so far:
   (a) Executing of the callback to inform the application about the new pool
   (b) Return value of the callback is either an accept or a veto.
7. **Backend:** Depending on the return value of the user function, either cleanup is done or the pool is added to the channel, and the reply sent to the assigning side.
8. **Backend:** The pool assignment reply is received and forwarded to the bulk library
9. **Bulk Transfer Library:** Depending on the outcome, the pool is added to the channel.
10. **Application:** The registered continuation gets executed, informing the application about the outcome of the assignment request.

It is important to note that the backend is responsible for cleaning up if the user application vetoed the assignment request. This includes removing the pool from the domain list if this pool was assigned for the first time.

Reasons for vetoing a pool assignment request may be that a pool has a wrong memory range or alignment, too small buffer sizes or other reasons from the application point of view.

5.2.4 Sending and Receiving

There are two operation modes of a channel. Depending on the role and direction, a channel can be seen to operate either in receive master or transmit master node. With the obligation for the master to provide buffers the two modes are slightly different:

- **Transmit Master:** Since, the master is on the transmit side, the buffers are already present in the sending domain and can be used for transfers. The receiving side may pass them back.
- **Receive Master:** In this configuration, the sending side is not necessarily in possession of buffers. Thus the receiver has to pass them first to the sending domain.

Note that this scenario is rather simple and it may be the case that the sender gets its buffers from another domain or allocates the resources by its own. The process of receiving or sending in the different modes can be seen in Figure 5.4. Without loss of generality, we explain the process at the move operation while the copy operation works analogously.

[7] Depending on the channel type and trust level

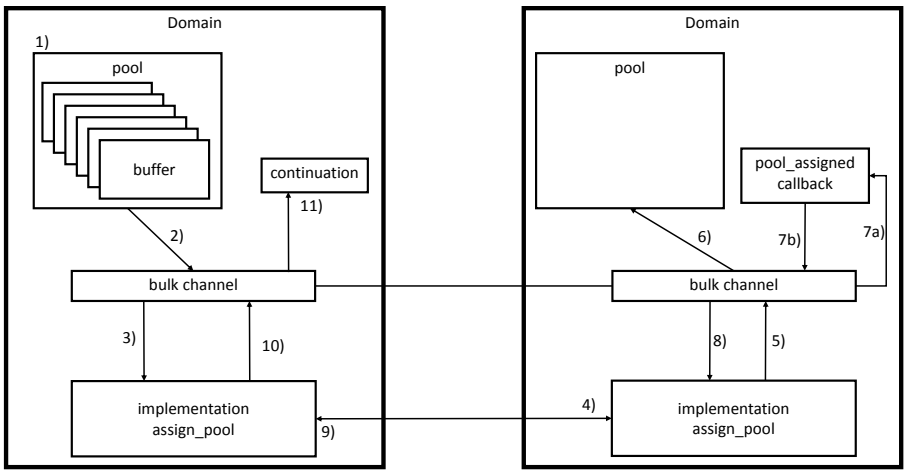

Figure 5.3: Pool Assign Operation

Transmit Master Mode

1. **Pool Allocation:** The transmit side allocates a pool
2. **Buffer Allocation :** The transmit side allocates a new buffer from the pool
3. **Move Operation:** The buffer state is changed in the transmitting domain and sent over the data channel to the receiver.
4. **Receive Event:** In the receiving side, the buffer state is changed, the `move_received` event is triggered and the application gets informed about the buffer.
5. **Pass Operation:** Again the state of the buffer is changed and passed on the control channel back to the transmitting side.
6. **Receive Event:** In the transmitting side, the state of the buffer is changed and the `buffer_received` event is triggered. The application returns the buffer back to the pool.

Receive Master Mode

1. **Pool Allocation:** The receiving side allocates the pool.
2. **Pass Operation:** The receiving side allocates all buffers and passes them to the transmitting side.
3. **Receive Event:** At the transmit side, the buffers states are changed, the `buffer_received` event is triggered and the application stores them in a suitable way.

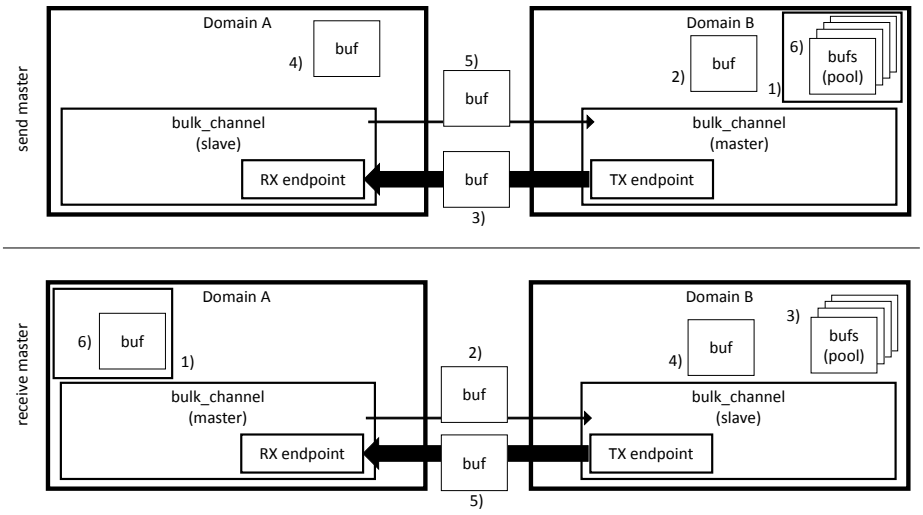

Figure 5.4: Bulk Transfer Architecture

4. **Sending Data:** The needed buffers is taken from the received buffers
5. **Move Operation:** The state of the buffer is changed and the buffer is moved over the data channel.
6. **Receive Event:** At the receiving side, the state of the buffers is changed and the `move_received` event is triggered. The application decides to return the buffer to the pool.

Chapter 6

Interface Specification

In the previous Chapters we have defined the terms and semantics of our bulk transfer infrastructure. In this Chapter we will outline a possible interface which implements the formal specification of the bulk transfer mechanism. First, we give an overview of the most important data structures (section 6.2), followed by the basic channel operations (section 6.3) and last the pool allocator (section 6.4).

6.1 Core Functionality

In order to use our bulk transfer infrastructure, we have written a library which provides a common interface to all backends. This library can be used by adding the following two inclusions to your code. Note that `<BACKEND>` has to be replaced by the backend you want to use.

```
1 #include <bulk_transfer/bulk_transfer.h>
  #include <bulk_transfer/bulk_<BACKEND>.h>
```

The `bulk_transfer.h` header file has to be included as soon as the bulk transfer library is to be used by the application. All the necessary declarations are contained within this single header file.

The backend specific header file `bulk_<BACKEND>.h` is currently needed to create the bulk endpoint which is required for the creation or binding process. See limitations (section 8.5) or future work (section 8.5) for further explanations on this topic.

6.2 Core Data Structure

Each part of the architecture as described in the previous Sections is reflected as a data structure in our library. We will explain the most important data structures in the following Sections. For a complete enumeration of all data structures please refer to the source files of our implementation.

6.2.1 Enumerations and Constants

As described in the formal specification (chapter 5) the possible states of channels or buffers and their properties are finite and clearly defined. Listing 6.1 on page 40 shows some of the used enumerations and constants. We want to highlight and explain some of them.

Channel Roles

The channel role enumeration contains three options. The `BULK_ROLE_GENERIC` can only be used on channel creation. This role is not valid when the channel is connected. If the creator is generic, its role will be adapted according to the choice of the binding side.

Trust Level

The interface provides a total of four different trust levels which are applied to pools and channels.

- `BULK_TRUST_UNINITIALIZED:` The trust level is not initialized. This only applies to pools that have not yet been assigned to any channel yet. This trust level is invalid for channels.
- `BULK_TRUST_NONE:` There is no trust on this channel. All security policies are applied to guarantee isolation.
- `BULK_TRUST_HALF:` An middle ground variant, that unmaps unowned buffers but does not revoke resources. Does not protect against malicious users, but against honest mistakes and accidents.
- `BULK_TRUST_FULL:` There is complete trust, all security policy changes are omitted.

Channel State

Each channel is in exactly one of the states defined in this enumeration. Some actions can only be done when the channel is in a specific state.

- `BULK_STATE_UNINITIALIZED:` This channel has not yet been assigned an endpoint i.e. the creation / binding procedure has not yet been executed.
- `BULK_STATE_INITIALIZED:` This channel is initialized i.e. the local endpoint is assigned. (Creator side only)
- `BULK_STATE_BINDING:` The remote endpoint has been assigned and the channel is waiting for a binding reply. (Binding side only)
- `BULK_STATE_BIND_NEGOTIATE:` The binding has been initiated and the channel properties are being negotiated.
- `BULK_STATE_CONNECTED:` The channel is fully operable.
- `BULK_STATE_TEARDOWN:` The teardown message has been sent. Messages in transit are received. No new messages can be sent.
- `BULK_STATE_CLOSED:` The channel is closed and the resources are freed.

Buffer State

When a buffer is copied, its state changes to read-only. In order to track which domain the first copy initiated from, we have introduced the buffer state `BULK_BUFFER_RO_OWNED`. This enables us to set the buffer to read/write again when the other copies have been released.

```
/** Specifies the direction of data flow over a channel. */
enum bulk_channel_direction {
    BULK_DIRECTION_TX, BULK_DIRECTION_RX
};

/** Specifies the endpoint role on a channel */
enum bulk_channel_role {
    BULK_ROLE_GENERIC, BULK_ROLE_MASTER, BULK_ROLE_SLAVE
};

/** trust levels of channels and pools */
enum bulk_trust_level {
    BULK_TRUST_UNINITIALIZED, BULK_TRUST_NONE,
    BULK_TRUST_HALF,          BULK_TRUST_FULL
};

/** channel states  */
enum bulk_channel_state {
    BULK_STATE_UNINITIALIZED, BULK_STATE_INITIALIZED,
    BULK_STATE_BINDING,       BULK_STATE_BIND_NEGOTIATE,
    BULK_STATE_CONNECTED,     BULK_STATE_TEARDOWN,
    BULK_STATE_CLOSED
};

/** represents the state of a buffer */
enum bulk_buffer_state {
    BULK_BUFFER_INVALID,  BULK_BUFFER_READ_ONLY,
    BULK_BUFFER_RO_OWNED, BULK_BUFFER_READ_WRITE
};
```

Listing 6.1: Bulk Transfer Enumerations

6.2.2 Bulk Channel

In our interface, the bulk channel is the central data structure. It contains the entire channel state such as trust level, direction or role. In addition to that, each channel is aware of the pools assigned to it. The complete representation of a bulk channel can be seen in Listing 6.2. We want to highlight some of the elements in the channel struct.

Callbacks

The implementation of the channel is event based. If the application wants to be informed about an event on the channel e.g. the arrival of a buffer, it can register a callback function which is called when the event occurs.

Constraints

There may exist some constraints on the channel such as memory alignment or range of supported physical addresses. This information is stored in the channel constraints and is used to validate the assignment of pools.

Implementation Data / User State

Each channel also supports additional data to be associated with the channel. The implementation data is intended to be used by the backend to store backend specific information. The application has the possibility to store additional state with the channel.

```
/** Handle/Representation for one end of a bulk transfer channel */
struct bulk_channel {
    /** callbacks for the channel events */
    struct bulk_channel_callbacks    *callbacks;
    /** the local endpoint for this channel */
    struct bulk_endpoint_descriptor *ep;
    /** the current channel state */
    enum bulk_channel_state          state;
    /** orderd list of assigned pools to this channel */
    struct bulk_pool_list           *pools;
    /** the direction of data flow */
    enum bulk_channel_direction      direction;
    /** role of this side of the channel */
    enum bulk_channel_role           role;
    /** the trust level of this channel */
    enum bulk_trust_level            trust;
    /** constraints of this channel */
    struct bulk_channel_constraints  constraints;
    /** the size of the transmitted meta information  */
    size_t                           meta_size;
    /** the waitset for this channel */
    struct waitset                  *waitset;
    /** pointer to user specific state for this channel */
    void                            *user_state;
    /** implementation specific data */
    void                            *impl_data;
};
```

Listing 6.2: Bulk Channel Struct

6.2.3 Bulk Endpoint Descriptors

The bulk endpoints are not represented explicitly with a data structure but are rather defined by an endpoint descriptor. Endpoint descriptors are backend specific and we refer to the backend specific part of the documentation or the source code for the endpoint descriptor specification.

6.2.4 Bulk Pool and Bulk Buffer

The bulk pools and buffers represent the registered memory usable for bulk transfers. Each buffer belongs to exactly one pool and the pool keeps track of

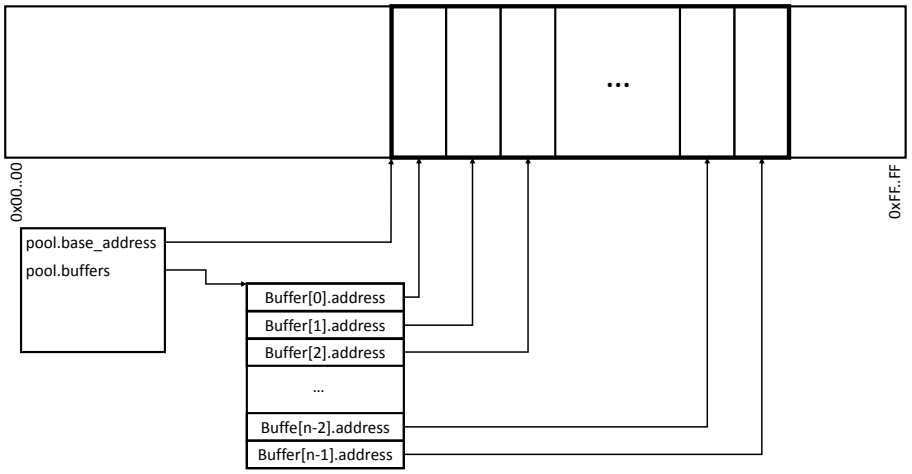

Figure 6.1: Pool and Buffer Representation

its buffers. Listings 6.3 and 6.4 show the complete definition. How the data structures are represented in memory can be seen in Figure 6.1.

Address Ranges

The virtual addresses as well as the physical addresses of the pools and the buffers stay fixed after the pool has been allocated. This enables the calculation of the buffer id from its address and the address from the buffer id.

Capabilities

Both the pools as well as the buffers contain a frame capability which represent the physical memory of the pool or the buffer respectively. The union of all frame capabilities of the buffers belonging to a pool will match the pool capability exactly. Depending on the trust level, the pool capability may not be present.

Trust Level

Like the channel, the pool also has a trust level. This is because we do not want to allow a pool being assigned to channels of different trust levels.

```
/**
 * The bulk pool is a continuous region in (virtual) memory that
 * consists of equally sized buffers.
 */
struct bulk_pool {
    struct bulk_pool_id     id;
    lvaddr_t                 base_address;
    size_t                   buffer_size;
    enum bulk_trust_level    trust;
    struct capref            pool_cap;
    size_t                   num_buffers;
    struct bulk_buffer     **buffers;
};
```

Listing 6.3: Bulk Pool Struct

```
/**
 * a bulk buffer is the base unit for bulk data transfer in the
     system
 */
struct bulk_buffer {
    void                    *address;
    uintptr_t                phys;
    struct bulk_pool        *pool;
    uint32_t                 bufferid;
    struct capref            cap;
    lpaddr_t                 cap_offset;
    enum bulk_buffer_state   state;
    uint32_t                 local_ref_count;
};
```

Listing 6.4: Bulk Buffer Struct

6.3 Bulk Channel Operations

Bulk channel operations can be divided into three fundamental categories. First, we have operations for channel management (subsection 6.3.2), second operations for pool management (subsection 6.3.3) and third the buffer transfer operations (subsection 6.3.4).

6.3.1 Asynchronous Interfaces

Before we explain the particular interfaces, we emphasize a common characteristic of all operations on the bulk channel. We decided to have asynchronous semantics for these operations. If the application wants to be informed about the outcome of the action, it can register a continuation so it will get an event when the action is finished. Listing 6.5 below shows the declaration of the continuation.

```
struct bulk_continuation {
    void (*handler)(void *arg, errval_t err,
                    struct bulk_channel *channel);
    void *arg;
};
```

Listing 6.5: Bulk Continuation

While all functions have an error code as return value, this error code is only based on local sanity checks e.g. sending a buffer over a not connected channel will always fail beforehand.

6.3.2 Channel Initialization and Teardown

The initialization semantics for a bulk channel are similar to those of sockets, where the socket corresponds to the endpoint and the call to listen corresponds to the `bulk_channel_create()` operation. On the other hand the `bulk_channel_bind()` operation corresponds to the call to connect. Listing 6.6 shows the function signatures for these operations.

```
errval_t bulk_channel_create(struct bulk_channel             *chan,
                             struct bulk_endpoint_descriptor *epd,
                             struct bulk_channel_callbacks   *cb,
                             struct bulk_channel_setup       *setup
                                 );

errval_t bulk_channel_bind(struct bulk_channel             *chan,
                           struct bulk_endpoint_descriptor *rem_ep,
                           struct bulk_channel_callbacks   *cb,
                           struct bulk_channel_bind_params *params,
                           struct bulk_continuation         cont);

errval_t bulk_channel_destroy(struct bulk_channel          *chan,
                              struct bulk_continuation      cont);
```

Listing 6.6: Bulk Channel Operations

Both the create and the bind function take parameters which specify the initial state values for the channel and are used to negotiate the final channel properties during the binding process.

Further, there is no continuation for the channel create operation. The creation does not involve another endpoint and thus only depends on the local domain. The result of the create procedure is directly signalled via the return value.

6.3.3 Pool Assignment and Removal

Recall that buffers can only be sent over a channel if the corresponding pool is assigned to that channel. Assigning a pool to a channel is a two way operation: only if the other side agrees to the assignment request can the pool be added to the channel. At assignment time, the necessary resources such as memory range and data structures are allocated. Thus it is important for an application to check the error value when the continuation is called (Listing 6.7).

Note that a pool can not be assigned to the same channel twice.

```
errval_t bulk_channel_assign_pool(struct bulk_channel       *chan,
                                  struct bulk_pool          *pool,
                                  struct bulk_continuation   cont);

errval_t bulk_channel_remove_pool(struct bulk_channel       *chan,
                                  struct bulk_pool          *pool,
                                  struct bulk_continuation   cont);
```

Listing 6.7: Bulk Pool Operations

6.3.4 Buffer Transfer Operations

The signatures of the actual transfer functions can be seen in Listings 6.8 and 6.9 respectively (each with their counterparts). All four functions take the buffer to be sent, the channel over which to send it and the continuation as arguments.

In addition to that, there is some meta data that can be transmitted along with the buffers. The meta data should be small compared to the buffer size e.g. a block id or request id.

The application has to be aware that when one of these functions is called on a buffer, the buffer may not be accessible to the application any more, or just in read-only mode in the case of copy.

```
errval_t bulk_channel_move(struct bulk_channel      *channel,
                           struct bulk_buffer       *buffer,
                           void                     *meta,
                           struct bulk_continuation  cont);

errval_t bulk_channel_pass(struct bulk_channel      *channel,
                           struct bulk_buffer       *buffer,
                           void                     *meta,
                           struct bulk_continuation  cont);
```

Listing 6.8: Bulk Move Operations

```
errval_t bulk_channel_copy(struct bulk_channel      *channel,
                           struct bulk_buffer       *buffer,
                           void                     *meta,
                           struct bulk_continuation  cont);

errval_t bulk_channel_release(struct bulk_channel       *channel,
                              struct bulk_buffer        *buffer,
                              struct bulk_continuation   cont);
```

Listing 6.9: Bulk Copy Operations

6.4 Pool Allocator

In contrast to the declarations above, the pool allocator is not part of the core interface. This decision is based on the observation that a domain can use the bulk transfer library without allocating buffers and pools on its own, receiving buffers from other domains instead. Therefore, in order to allocate a new pool of buffers, we need to include the bulk allocator header:

```
#include <bulk_transfer/bulk_allocator.h>
```

The following Sections will briefly describe the most important functions of this header that are needed to allocate a new pool.

6.4.1 Allocator Initialization

Initializing an allocator will create a new pool with the corresponding bulk buffers. The number and size of the buffers are given by the arguments. Listing 6.10 shows the function signature for the initialization.

It is possible to initialize the allocator with additional constraints. These constraints can set the possible memory range for allocation, the pool trust level or special alignment constraints for the buffers.

```
errval_t bulk_alloc_init(struct bulk_allocator        *alloc,
                         size_t                        count,
                         size_t                        size,
                         struct bulk_pool_constraints *constraints)
                             ;
```

Listing 6.10: Bulk Allocator Initialization

When the allocator initialization is completed, the memory for the pool is allocated and mapped in the domain.

6.4.2 Allocating and Returning Buffers

As soon as an allocator is initialized, it can be used to get bulk buffers. Listing 6.11 shows the signatures of the buffer alloc and free functions.

Notice that the buffers are in fact always there, the allocator only tracks the unused buffers.

```
struct bulk_buffer *bulk_alloc_new_buffer(
                                    struct bulk_allocator *ac);

errval_t bulk_alloc_return_buffer(struct bulk_allocator *alloc,
                                    struct bulk_buffer    *buffer);
```

Listing 6.11: Bulk Allocator Initialization

6.5 The Bulk Transfer Library

Recall that the generic part of the bulk transfer library provides a unified interface to the different backends as described earlier in this chapter. Under

the hood, the library also provides common functionalities that are used by the different backends. We want to give an overview of these functionalities as they are useful when developing a new backend.

6.5.1 Contracts of Public Interface Functions

The application calls the backend functions via the public interface as described in chapter 6. These public interface functions ensure that the preconditions stated in chapter 5 are satisfied before invoking the backend specific function. Further, the buffer state is changed according to the invoked operation – this implies that the backend does not need to deal with buffer state changes and that the buffer contents may not be accessible any more[1].

When getting invoked, the backend can assume that this operation is valid with the given parameters. In addition to that, the backend may introduce additional checks related to backend specific state e.g. if there is space left in the transmit queue and return a proper error code if not.

6.5.2 Buffer Related Functions

There are functions available to query the buffer state, to check e.g. if a buffer is owned by the domain or if that buffer is a copy. The bulk transfer library provides functions for mapping, unmapping and changing the protection rights of a buffer. These functions are accessible from the backends. These operations are no-ops if the channel is trusted and involve a page table modification otherwise.

We provide a single function that does the corresponding map/unmap/protect instruction depending on the state transition the buffer is about to make. The signature of this function is shown in Listing 6.12.

```
1 errval_t bulk_buffer_change_state(struct bulk_buffer *buffer,
                                    enum bulk_buffer_state new_st);
```

Listing 6.12: Changing a Buffer State

6.5.3 Pool Related Functions

Managing pools and tracking them is one of the core functional requirements of the bulk transfer library. We provide a set of functions that

- Compare two pool ids and generate a new unique one
- Track the pools assigned to a channel and check if the pool has been assigned to the channel
- Allocating pool data structures
- Mapping / Unmapping of pools

[1] In the untrusted case the buffer may be unmapped

Domain Pool List

As specified in chapter 5, we allow each pool to only be present once in the domains. To ensure this, the library maintains an ordered list of pools present in this domain. The policy of this domain pool list is the following:

- **Allocators:** When allocating a pool, the allocator must make sure that this pool is inserted into the domain pool list.
- **Backends:** Upon receiving a pool assign request, the backend must make sure that a pool is not created twice.

Listing 6.13 shows the interface to the domain pool list. If a pool is not present in this domain, the getter function will return `NULL`.

```
errval_t bulk_pool_domain_list_insert(struct bulk_pool *pool);

struct bulk_pool *bulk_pool_domain_list_get(
                                    struct bulk_pool_id *id);
```

Listing 6.13: Interface to the Domain Pool List

Chapter 7

Application

As a use case of our bulk transfer infrastructure, we implemented a basic block server which makes use of the different backends. The application setup can be seen in Figure 7.1. This basically establishes a two hop bulk channel with different backends.

7.1 Domains

As one can see, the setup consists of tree different domains running on two different machines.

7.1.1 Network Block Service (NBS)

The network block service domain is located on a different machine than the other two domains. This domain is responsible for managing the blocks located on this machine. In our implementation this block store is just a distinct region in physical memory.

The domain exports two interfaces:

- **Service Interface:** This interface exports the functionality of the block service and is used to issue read requests, channel initialization and status messages. The service layer is implemented as a TCP server.
- **Bulk Interface:** There are two bulk channels opened during the connection process. The actual data transfer of the blocks is going over these channels.

7.1.2 Network Block Service Client (NBS Client)

This domain serves as a network client for the block service and does not store the the blocks locally[1]. There are two modules in this domain:

- **Network Client:** This is the counter part to the network server located in the network block service. When the domain is spawned, the network client initializes the connections to the network server.

[1] There is no caching available at this time

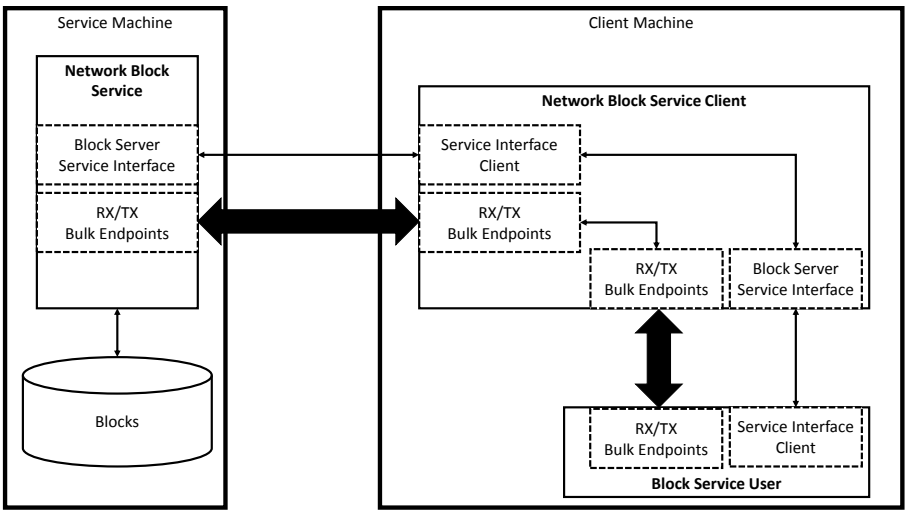

Figure 7.1: Network Proxy using Local Channel

- **Machine Local Server:** This module provides the block service interface to other domains located on this machine. The interface is implemented as a Flounder service.

The main purpose of this domain is to forward requests and messages received on the local server over the network client to the network block service and vice versa.

7.1.3 Block Service User (BS User)

The block service user domain issues read/write requests of blocks and sends them to the local service interface of the NBS client.

7.2 Connection Initialization

The connection setup between the NBS and the NBS client is done using a tree-way handshake protocol. When the block service starts, it starts listening on a well-known port.

1. **Connection:** Client connects to the TCP server of the block service and waits until the connection is established.
2. **Setup:** Client creates the two bulk channels and sends the endpoint information to the server.
3. **Bulk Binding:** The NBS binds to the channels. As soon as the binding is complete, the NBS client exports the Flounder interface.

This connection setup procedure is adapted to the Flounder interface accordingly when the BS user connects to it.

7.3 Pool Assignments

Pool assignments are initiated by the BS user domain, which allocates the pools and assigns them to the channels. The NBS client will eventually receive a pool assignment request and forward it to the NBS. The pool assignment process is finished when the pools are assigned over both channels.

7.4 Serving Requests

There are two types of requests that can be executed on the block service.

Read The read request is sent over the service layer and has support for batch requests. To send the bock, a bulk buffer is allocated and the data copied from the block into the bulk buffer before sending.

Write The write request is sent directly over the bulk channel. There is no support for batch writes. The buffer contents are copied from the bulk buffer into the corresponding block.

7.5 Basic Work Load

As soon as the pools are assigned to the channels, the BS user starts issuing requests to the block server. As a basic workload, the client issues a write request by doing a bulk move operation and waits for the acknowledgement (status message). Then it issues a read request, waits until the data arrives and then checks the data for integrity. This sample workload is just for demonstration of correctness.

Chapter 8

Conclusion

8.1 Support for Variable Sized Buffers

The current implementation does not support sending variable sized buffers. All the buffers belonging to the same pool must have the same size. Depending on the backend, this may result in quite an overhead, when sending buffers that are just half full.

A possible way of doing this would be to introduce the length information in the buffer and provide an interface like Java byte buffers [17] to read or write the data.

8.2 Enforcing Policies

With the capabilities currently implemented in Barrelfish, a clean enforcement of the security policies is not always possible. When a pool is allocated there will be a master capability for this pool. Therefore, the allocator of the pool can map the pool at any time at another memory location and mess with the data in the buffers. One way to overcome this issue, would be to have a third trusted domain which is responsible for allocating pools and just hands the buffer capabilities out to the domains using the bulk transfer facility. A different approach would be to use a new type of capability, as it will be introduced in the shared memory backend report.

8.3 Device Drivers

Many devices, especially block devices like hard drives, deliver a significant amount of data, have DMA support and are making use of equal sized buffers.

Adapting existing device drivers for disks or the USB stack to support bulk buffers as backing memory would be a valid option to consider. Building a file system server which makes use of the block device driver or a usb mass storage driver and the bulk transfer framework would be an interesting use case to investigate.

8.4 Aggregate Objects

As already explained in Section 4.4, having the possibility to send aggregate objects would simplify the transmission of larger data blocks at once (from an application point of view). Further having a dynamic aggregate object data structure also enables to insert / remove blocks of memory within the object.

As an addition, for simplified access the aggregate objects library may provide the possibility to map the aggregate object as a single contiguous range of virtual memory. This possibility will lead to the interesting question on how this would be conform with our formal specification of the bulk buffers: If the buffer is mapped as part of an aggregate objects it should not be possible to send it further, otherwise the data of the buffer could be corrupted by writing to the mapped aggregate location.

8.5 Name Service Integration

With Flounder [3] the domains can export their `iref` and associate it with a name in the nameservice, to enable other domains to look them up based on the service name. A similar approach may also be beneficial with the bulk transfer endpoints. However, for some endpoints a single `iref` value is not enough to fully represent the endpoint.

As an endpoint may only be used by at most one channel anyway, it makes more sense to have an orthogonal service that the domain exports, which creates the requested endpoints on the fly. Currently, this is the approach we have implemented in our block service application, where we use the block service channel to exchange the endpoint information.

8.6 THC Integration

We have chosen to design and implement the bulk transfer mechanism based on asynchronous events and waitsets. This design might be easily adaptable to be integrated into THC [13]. This integration can lead to a unified way of exchanging messages of different size between domains and even machines.

Part II

Network Bulk Transfer

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the distributed architecture of today's computer systems the user's data is not just kept on a single machine but is rather stored on multiple machines in a local network or even on the Internet. The local machine serves as a cache for frequently accessed files. Storing the files on a remote machine increases the need for high bandwidth communication to transfer the requested data in an acceptable time to the client machine.

Data processing may be further distributed among different processes or even different machines. Once the data arrived on the client machine, the copying of data should be avoided whenever possible. Applying the zero copy principle does not only reduce the work for doing the actual copy but also lowers the memory footprint because the data is only present once. Therefore, a seamless integration of the bulk transfer mechanisms into the networking framework is required. In other words, the bulk transfer facility must have a network backend.

As one of the main goals of any bulk transfer implementation the avoidance of copying data cannot fully be applied to the network: The data will be copied by the network card during the transmission process. In any case, the network bulk transfer implementation should be designed in such a way that the data can be sent or received directly out of the bulk buffers used on the local machine.

The fact, that there is a real copy needed with every operation results in a slightly different problem description compared to other backends. For instance, compared to the shared memory backend, data integrity must not be enforced as the sending application cannot write into the memory of the remote machine[1]. However, another problem arises: there must always be enough receive buffers available in the receive queue of the network card. Otherwise arriving data cannot be received.

Report Outline In this report, we will first give an overview of related work (Chapter 2) followed by a brief explanation about the background of networking and Barrelfish (Chapter 3). The next chapter we will describe our design choices and implementation details (Chapter 4). The last two chapters of this report are about the evaluation of our implementation 5 and the conclusions drawn from the evaluation 6.

[1] Assuming no remote direct memory access

Chapter 2

Related Work

This chapter provides an overview of other research work related to different aspects of our project.

2.1 Transferring Bulk Data over Network

When thinking about moving larger amounts of data over a network, the obvious choice is to use the default socket based network API. But this API is not particularly well suited for transfers with high performance requirements [12]. Thus high performance computing (HPC) applications often use different interfaces, remote direct memory access (RDMA) being one of the more popular abstract interfaces. Its main benefit is the availability of one-sided operations and the ability to specify the destination buffer on the sending side. There are a number of RDMA implementations including Inifinband and also over Ethernet such as RDMA over converged Ethernet [5]. Portals [7] is another interface that is designed for low overhead data transport over different technologies from off-the-shelf Ethernet to proprietary interconnect technologies, and also provides a shared memory implementation.

2.2 Transport-agnostic Message Passing

In the context of message passing systems the idea of providing an interface that can be implemented over a wide range of transport mechanisms is well known. MPI is a standardized interface that is widely used in HPC projects. OpenMPI [11] by Graham et al. uses a component based architecture to provide flexibility and supports communication over shared memory as well as a number of different networking technologies. The Barrelfish messaging infrastructure based on Flounder [3], also has a similar goal of allowing different underlying mechanisms to be used. While Flounder was initially only used for communication within one machine, Hauenstein et al. [14] extended it by an Ethernet backend, that allows for communication across machine boundaries.

2.3 Efficient Send/Receive of Packets

The fact that an API that requires expensive operations in the data path for sending and receiving packets can significantly impact the overall networking performance has been recognized early on, and since then there were multiple attempts for designing more efficient send and receive paths. Von Eicken et al. with their U-Net project [24] were among the first to propose direct interaction of applications with the NIC, based on ATM NICs. Applying similar ideas to Ethernet is a bit more challenging since there the problem of how to demultiplex packets into different hardware queues cannot be easily solved on L2, since Ethernet provides packet semantics and thus no flow identifiers akin to the ATM circuit identifiers are available. This issue was tackled by Pratt and Fraser [18] based on the Arsenic Gigabit Ethernet NIC and the idea of using filters to steer packets to queues. The main issue with these approaches is generally that they require specific hardware providing the respective functionality, and that the code for accessing the queues is highly NIC-specific. This lead Rizzo to develop netmap [20], a system for high throughput packet processing that does not rely on specific hardware and does not require changes to applications.

2.4 Use of Hardware Features

Antoine's bachelor thesis on low-latency networking [16] is the source of the Intel 82599 driver in Barrelfish, and showed that the effective use of NIC hardware features can significantly decrease latency. This work was later extended to investigate the impact of hardware configuration on general networking performance by Shinde et al. [23], showing that while hardware features can provide significant performance improvements, finding an optimal configuration for a particular scenario is non-trivial. Huggahali et al. [15] quantified the performance characteristics of direct cache access. The benefits of jumbo frames for 10Gbit Ethernet were analyzed by Feng et al. in the context of their performance analysis of a TCP offload engine [10].

Chapter 3

Background

3.1 Intel 82599 NIC

We decided on focussing on the Intel 82599 10-Gigabit Ethernet card [8] for our implementation because of the availability of a Barrelfish driver for it and prior experience. In addition to basic Ethernet functionality the card provides an interesting set of Hardware features that can be exploited to achieve better performance or even allow for different designs. In the following section we will discuss a set of selected features:

3.1.1 Checksum Offload

Hardware checksum offload is one of the most basic offload features. Ethernet CRC calculation and CRC check have been provided even by cheap 100Mbps Ethernet NICs such as the Realtek 8139. In addition to Ethernet CRC offload, the Intel 82599 provides the following checksum offloads both for sending and receiving: IPv4/6,UDP,TCP (and SCTP). Note that the TCP/UDP checksum offloads in the send path are not complete, and require the network stack to calculate the checksum over the pseudo-header in advance and store it in the checksum field before passing the packet to hardware.

3.1.2 Jumbo Frames

Another common hardware feature is support for jumbo frames, i.e. Ethernet frames longer than the default of 1500 bytes, thus allowing for larger packets to be sent. The Intel 82599 supports jumbo frames of a total length of 15.5kB. Note that jumbo frames need to be supported in all components involved (sender, receiver, and switch).

By allowing larger frames, the number of frames for sending a fixed amount of data is reduced, which in term implies fewer interactions with the NIC on both ends and thus a lower overhead. Jumbo frames can significantly improve throughput [10].

3.1.3 Queues

Multiple hardware queues are available on most high-end NICs and increasingly common on cheaper NICs. They can allow for improved performance and better quality of service (e.g. isolation of flows). The Intel 82599 provides 128 send and receive queues. The interaction with these queues is based on memory mapped descriptor rings and two index (head and tail) registers per queue.

For the send side, using them primarily reduces the need for synchronization when sending from multiple cores, and can isolate individual flows, depending on the scheduling algorithm used by the card for sending out packets from multiple queues.

On the receive side, multiple queues make it easier to distribute incoming packets to multiple cores, without the bottleneck of a single core fetching and demultiplexing packets. Another benefit is the possibility of placing packets in different buffers depending on the queue, which could be used to place packets in memory from the NUMA domain of the processing core, or to place the packet data directly in application-specific buffers and thus avoiding copies.

3.1.4 Filters

In order for multiple receive queues to be useful, a mechanism for steering packets to queues is required. The Intel 82599 provides a number of different filters, including 128 5-tuple filters: These allow steering packets based on an ordered set of filters, where each filter can match on the protocols used and source/destination IP and port with the possibility of specifying a wild-card for each field. There are a number of other filters supported by this card, but these are beyond the scope of this report.

3.1.5 Header Split

For certain applications it is desirable to receive the packet payload (e.g. TCP payload) in a different buffer than the headers. A possible application for this might be that data should be received directly into application buffers while the headers stay in kernel buffers. This implies that the payload itself will be properly aligned in the application buffer, which may simplify further processing. With the Intel 82599 this can be achieved using the header split feature. It can split Ethernet, IP, UDP, TCP, SCTP, and even NFS headers.

If header split is disabled, each descriptor in the receive queue only contains the address of one receive buffer of a size that is fixed per queue. Enabling header split means that each descriptor will now reference two buffers, a header buffer and a data buffer, with the size of the header buffer being configured separately per queue. In addition there is a configuration register specifying which headers should be split (the card tries to split as many headers as possible, going from lower to higher layers). When receiving a packet and its header can be split, the header will be placed in the header buffer and the length of the header will be reported in the descriptor. If no header can be split, the header buffer will be left empty (assuming that the "always split" configuration option is disabled, otherwise the card will first use the header buffer and then the data buffer).

3.1.6 Direct Cache Access

When a packet arrives from the Intel 82599 usually the cache coherency mechanism will make sure that the respective cache lines are invalidated, and thus the CPU will see a coherent version when looking at the buffer. But this also means that accessing a buffer for a newly received packet will inevitably lead to cache misses. Direct cache access allows the NIC to directly write the packet into the cache of a specific core, and thus prevent these cache misses. This can lead to significant improvements in latency [15], but can also have an adverse effect because of cache thrashing if there are many packets arriving that are not processed immediately.

3.1.7 Virtualization

Another interesting feature of the Intel 82599 is virtualization support using PCI SR-IOV. If virtualization is enabled, the card will present a configurable number of virtual PCI functions. The hardware queues are divided among virtual functions, and each virtual function only allows access to its assigned queues. Note though that not all filters can be used in combination with virtualization.

3.2 Barrelfish

This section provides a brief outline of the current Barrelfish network subsystem at the time of writing this report.

3.2.1 Network Stack Concept

The system is based on an Exokernel approach, where the goal is to push the processing to the application as much as possible, leading to a more flexible system and better performance.

Network Daemon: `netd`

`netd` provides the network services that cannot be implemented on a per application basis and will handle all packets not directed to other applications. Initialization of the network stack using DHCP or from a static configuration is one of the services provided. Second, ICMP requests (such as ECHO) will be handled by `netd`, also it will send out ICMP destination non-reachable messages for packets not directed to an application and not intended for itself. In addition `netd` is responsible for responding to and sending out ARP requests.

Network Device Manager: `NDG_mng`

The network device manager handles requests from applications to allocate unused port numbers or to direct traffic with specific destination port numbers to the application. The latter depends on the respective driver, and could result in an additional software filter, or a hardware filter being configured by the driver.

Network Card Driver

A network card driver is responsible for initializing the network card, and generally interface with the network card. Depending on the hardware this might be restricted to control plane operations such as initializing the card and registering hardware filters. For other cards the driver is also involved in the data path, for receiving and sending packets.

Application

While the Exokernel design allows applications to directly process their network packets, most Barrelfish applications use the default lightweight IP (lwIP) based protocol stack. lwIP provides both a POSIX socket interface as well as an lwIP specific interface that gives the application more control.

3.2.2 Intel 82599 Support

In Barrelfish the Intel 82599 is referred to as `e10k` (in analogy to the `e1000` driver for Intel 1Gbps NICs). The `e10k` driver is split up into two separate domains: `e10k_cdriver` performs initialization of the hardware and is responsible for registering filters to steer packets to queues on requests from the device manager. `e10k_queue` provides access to one hardware queue (multiple instances need to be started if multiple queues are to be used), and provides the generic Barrelfish queue manager interface, that is used by applications to send and receive packets. The `e10k` drivers implement a wide range of hardware features. For this project support for jumbo frames and header split was added.

While the Intel 82599 does not support safe userspace networking without virtualization being enabled (since no IO-MMU can be used to protect memory), the infrastructure does allow unsafe userspace networking where one or more NIC queues can be directly mapped into an application. This is desirable since direct access to the queues in the application reduces the communication overhead that would otherwise be needed for communication with the queue driver.

3.3 Security Aspects and Encryption

Ensuring data integrity and confidentiality plays an important role especially when data leaves the machine. By using networking technology the data may even leave the local network and reach its destination over the Internet.

In our analysis of the options we will completely ignore any security aspects such as data encryption since this is beyond the scope of this course. The responsibility of ensuring data integrity and confidence is left to the application programmer who can encrypt the data before it is written to the buffer.

Just to mention an alternative it would be an option to provide another backend, which makes use of one of the available security extensions such as SSL or IPSec. When an application programmer uses that security enabled interface it is transparent to the programmer that there will a certain overhead introduced.

Chapter 4

Implementation

In this Chapter we briefly describe the most important points of our network backend implementation. First we start with a Section describing the ideal network implementation. Secondly we relate this ideal networking to reality and outline the arising difficulties. Third, we present the implementation decision and their rationale. The last Section consists of an overview of different implementation variants.

4.1 Ideal Implementation

A first step towards an implementation of a network backend for the bulk transfer mechanism is to identify the characteristics of an ideal implementation.

4.1.1 Performance

One of the first goals that comes to mind is good performance, so the backend should reduce the overhead of communication compared to the underlying network mechanism to a minimum. By performance we are generally interested in latency and throughput. Here the latency is the time delay from when an operation is issued on one side until it is received and processed on the other side. And the throughput refers to the number of operations that can be performed over a specific time frame, e.g. when sending move operations over a 10Gbps network the goal is to be able to reach a throughput close to 10Gbps for transferring the actual data.

4.1.2 Transparency

Another major goal for this project is *transparency*, i.e. an application using the bulk transfer mechanism should not need to know what backend it is running on. One implication of this, that it is not immediately obvious from the specification, that an application might rely on the fact that there is a correspondence between memory ranges on both sides representing a single buffer.

For example when an application at an endpoint writes to a buffer for which it previously received ownership over this channel and moves it back over the channel, the receiving side could rely on the fact that it received the data in the same buffer it previously passed. This could be relevant for applications relying

on specific characteristics of the buffers, such as e.g. the physical address if they are doing I/O and are providing the other side with some information about which buffers to use for what.

4.1.3 Reliability

A useful implementation should provide some reliability guarantees, i.e. ideally we would like all operations to succeed assuming valid parameters are provided. This means that the backend needs to provide protection against lost and corrupted packets and handle these problems internally invisible to the user. Note that there are network failures that cannot be hidden in software, such as e.g. a broken physical link, in which case the operations should fail gracefully.

4.2 Difficulties in Reality

After identifying the goals and characteristics of an ideal implementation, we can start thinking about how to construct a real implementation and figure out what problems and constraints are imposed by the given hardware.

4.2.1 Transparency vs. Performance

One point to notice is that stronger transparency guarantees require more communication than weaker ones, thus leading to a performance vs. transparency trade-off. On one extreme there is a fully transparent channel, where basically each operation has to be relayed to the other end, such as adding a buffer, or passing ownership for a buffer. If we are willing to drop most transparency guarantees we can come up with a scheme that only requires network interaction for the actual data operations (i.e. move and copy), but requires the receiver to allocate its own buffers and provide them to the backend and gives the sender no control over which buffer the data will end up in (this corresponds to the proxy implementation described bellow).

4.2.2 Reliability vs. Performance

Another trade-off exists between reliability and performance. Note that this is a general and well-known problem for networking applications, and the usual way to handle this is to use different protocols (such as TCP and UDP) that provide a different set of reliability guarantees, and applications can decide on what to use, based on their requirements.

4.2.3 Copies

What we ideally would like to have when sending a buffer over the network (e.g. by doing a move operation) is that the data will directly be stored in the right buffer by the NIC. This would omit the requirement for additional copying of data, since copying data around implies additional CPU cycles which are used to process the receive event. There are basically two ways to achieve this:

Either using an intelligent NIC that can actually select the right buffer based on the received packet. Note that this requires significant hardware support, and is similar to an RDMA write, with the difference that we do not need the

operation to be one-sided. In fact we actually need a notification to let the receiver know that it received data in this specific buffer. But the only way to control the destination buffer with our Intel 82599 is basically to use different hardware queue for every buffer and let the hardware demultiplex the incoming packets e.g. based on the destination port, but in any realistic use case the 128 hardware queues of the Intel 82599 would not be sufficient.

The other possibility to avoid a copy is to send a packet ahead specifying the destination buffer for the next packet, allowing the receiver to set up its receive queue accordingly by making sure that the destination buffer will be in the first position on the receive queue. First off, note that this basically implies an additional round-trip for every move/copy operation, since the sending side needs to wait for an acknowledgement. This also means that data operations need to be sent sequentially (or batching is needed to include multiple destination buffers in the announcing message). Note that the overhead imposed here, will almost certainly outweigh the cost of a copy operation in software. Further, there need to be a guarantee that no other message arrives at the receiver after the queue has been setup. This would require a lock on the channel which is something we want to avoid.

4.2.4 Packet Processing Costs

Previous benchmarks have shown that our hardware allows round-trip times around 10 μs for small packets between two machines. While fast communication is obviously desirable, this also means that software processing time for packets is no longer dominated by propagation delays in the network, and thus the software path also has to be carefully optimized in order to get good performance.

4.2.5 Performance $\neq$ Performance

Another difficulty is that optimizing for one aspect of performance can often negatively impact other aspects. An example of this is the use of interrupts for signalling packet arrival: In order to get a minimal latency, it is desirable to get interrupts as early as possible, and avoid interrupt throttling. For improving throughput on the other hand, interrupt throttling is desirable, since it reduces the overhead of interrupt processing, and received packets can be batched. Direct cache access (DCA) is another feature that has similar performance characteristics: if there are relatively few requests that are to be handled with minimal latency, enabling DCA for a queue makes sense since it will make sure that packets are in the cache before processing starts, thus saving cache misses. But if a lot of requests are arriving concurrently, DCA can lead to cache thrashing and thus negatively impact performance of other code running.

4.3 Implementation Decisions

Based the analysis of possibilities in the previous section, we have a clear understanding what is achievable given our hardware. The question at hand is now is how to make the best out of this situation and make optimal use of

the hardware. This section outlines some of our implementation decisions and provides a rationale for them.

4.3.1 Multiple Implementations

Because of our observations in the previous section, that there is a clear trade-off between transparency and performance, we decided to implement multiple versions of a network backend each providing different semantics. Thus allowing the user to choose the appropriate backend depending on the required transparency guarantees and the desired performance. We ended up implementing 3 different backends (detailed descriptions are provided in the next section):

- Proxy (see 4.4.1)
- Fully Transparent (see 4.4.2)
- No Copy (see 4.4.3)

4.3.2 Directly Mapping Hardware Queues

In order to achieve better performance and better flexibility we decided on mapping the used hardware queues directly into the application address space. Note that the Intel 82599 does not support safe user-space networking without virtualization enabled and the use of an IO-MMU. However, this is a problem that can be rectified by using hardware that provides better support for user-space networking, or by relying on virtualization.

The alternative to user-space networking would be to rely on the `e10k_queue` driver for accessing the queue, which would mean additional overhead for communicating with the driver for every send and receive operation. Note that using a directly mapped receive queue means that the hardware needs to be able to steer packet intended for our bulk transfer endpoint into our queue (more about this below).

4.3.3 Alignment of Received Payload

Since one of the goals identified previously, we need a way to ensure that the buffer-data part of an incoming packet will be copied by the card into the buffer, starting at the beginning of the buffer. Since the backend implementation does not control how the buffers are allocated, we cannot avoid this problem simply by allocating larger buffers, and pointing the receive descriptors to a modified address.

The first obvious difficulty is posed by the protocol headers that precede the payload of the packet. But this can be easily addressed by using the Intel 82599 header split feature, causing the headers to be placed in separate buffers (but this only works for headers of the standard internet protocols such as TCP/IP or UDP).

Another problem is that the buffers we intend to use are larger than the 1500 bytes that fit in a standard Ethernet frame [1]. The usual way to handle this issue is to rely on fragmentation/segmentation. But this leads to another

[1] Currently the implementation constrains the buffers to be at least page size and a power of 2

Ethernet Header	IP Header	UDP Header	Buffer	Meta Data	Ethernet CRC
14 bytes	20 bytes	8 bytes	4096 or 8192 bytes	X bytes	4 Bytes

Table 4.1: The Transmitted Packet Format

problem since the payload will now be split over multiple packets. Just doing the math and adding different offsets to the buffer address and storing it in multiple successive receive descriptors, only works reliably if we have a guarantee that the segments/fragments arrive in order and there are no other packets arriving in between, which is not feasible in practice. Our solution here is to rely on jumbo frames, which allows us to send buffers of size up to 8192[2] bytes (assuming a size of a power of 2) in single Ethernet frames.

The last issue is that in addition to the buffer data, we generally need some control data, such as a message type and some meta data. Now if we add this information as a an additional bulk transfer specific header after the default protocol headers, we destroy all our efforts up until now to get the data nicely aligned into memory: Only the standard headers can be stripped by the NIC and thus our additional header will be written to the begin of the buffer. To avoid this, we decided on using a trailer and storing the control data after the buffer data. Meaning that it will be stored in the buffer specified by the next receive descriptor. Assuming we are using a packet-based protocol that provides the packet length in its headers it is also easy to find the trailer and identify the message type for variably sized messages by looking at the last byte, which specifies the message type. Table 4.1 shows the resulting packet format.

4.3.4 Protocol Choice

We decided early on against relying on the Barrelfish default lwIP network stack, mainly for performance reasons and since it seriously reduces control for us about how buffers are used for sending and receiving data over the network, compared to interfacing with the NIC directly.

While TCP might seem like a reasonable choice at the first glance, there are difficulties: First off, TCP is very complex which makes it hard to implement correct while still delivering good performance. Another problem is that TCP is stream based, which means that we do not have any control on how the payload will be split up into Ethernet frames, which also means that receiving directly into the buffers is not possible.

This basically leaves us with Ethernet, IP and UDP to choose from. We ended up deciding on UDP, mainly because it allows for easy demultiplexing of the packets into the respective hardware queues, using 5-tuple filters that use the UDP destination port. Note that we could also use IP packets and demultiplex based on the IP address using 5-tuple filters, but this is less practicable and does not really offer any advantage over UDP (besides the few bytes saved for the header), since UDP does not require any protocol processing besides stripping the header and thus also adds no significant overhead. Directly using Ethernet frames would make demultiplexing harder, since the 5-tuple filters cannot be used there, and the only possibility would be to use one of the 8 EtherType

[2]The maximum supported by the Intel 82599 is actually 15.5kB

filters, to demultiplex based on the protocol type, which severely limits the number of queues that can be used.

4.3.5 Prepared Headers

On the send path, the headers involved, Ethernet through UDP, are almost static. The only fields that potentially change with different payloads are the IP and UDP length fields and the checksums. This allows us to prepare a header buffers for each slot in the transmit queue at binding time, and at the time the buffers are used, only the length fields need to be modified (the checksums are offloaded to hardware). Not having to fill in the full header for each send operation saves some CPU cycles.

4.3.6 Reliability

Note that our current implementation does not provide any mechanism for flow control or handling lost/reordered packets. Since we are mostly doing benchmarks using machines that are connected back to back, this is not an issue for our benchmark, but needs to be addressed for practical applications. We skipped this for the sake of simplicity and due to time constraints.

4.4 Channel Variants

There are several options to implement the network backend of the bulk transfer infrastructure. Based on the insights gained while analysing the possibilities in reality we have implemented three different approaches each having its own benefits and tradeoffs.

4.4.1 Network Proxy

As the name suggests, we have a distinct network endpoint which serves as a proxy for sending or receiving over the network. This endpoint basically just listens for data arriving on the bulk channel and forwards them to the network card or vice versa.

Overview

This particular endpoint can either be a separate domain as shown in Figure 4.2 or in a thread running inside the domain (Figure 4.1). The only thing that changes is the channel type that connects to the network proxy endpoint.

Using the proxy variant to connect two machines with each other we end up in having virtually one channel between the client machine and the service machine, However, there are in fact tree channels under the hood. Each bulk buffer operation makes three hops till it gets to the destination.

Channel Semantics

The two proxy variants do not fully confirm the channel semantics as described in the formal specification. Both ends of the virtual channel need to be the master of their sub channels to the network endpoints. This implies as a channel

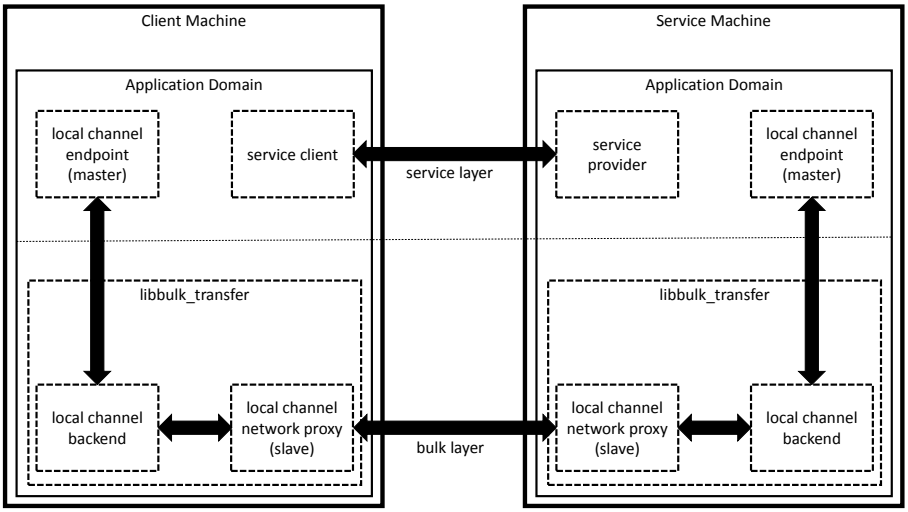

Figure 4.1: Network Proxy using Local Channel

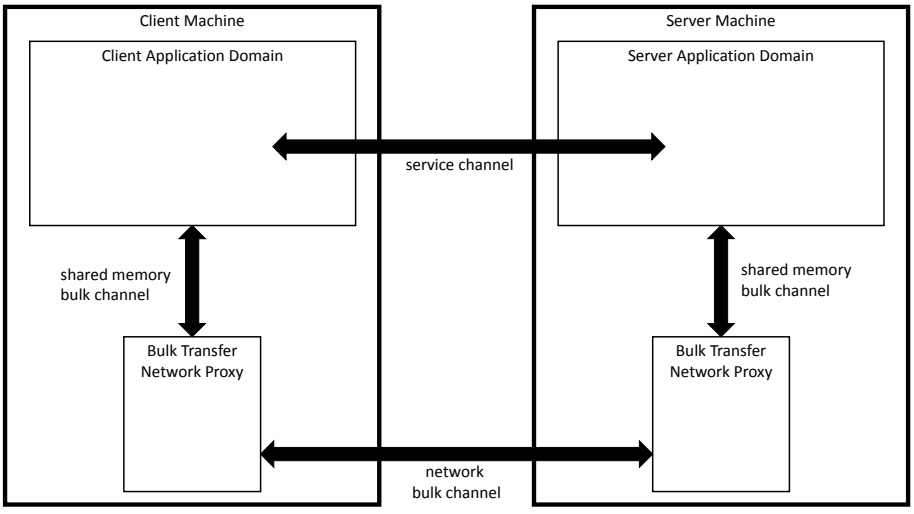

Figure 4.2: Network Proxy using Shared Memory Channel

master, both sides need to provide buffers to the channel in order to receive data. While the role semantics apply to the two sub channels, at the global view of the virtual channel this does not apply since we virtually have two endpoints being in master role.

Initialization

On both machines the channel initialization is done similar and involves basically setting up a channel to the network endpoint located within another thread (using a local channel) or on another domain (using a shared memory channel). In the following initialization example we will use a local channel to demonstrate the steps:

1. Creation of a local endpoint of type local channel
2. `bulk_channel_create()` with the local endpoint and the role `BULK_ROLE_MASTER`
3. Creation of the remote endpoint of type local channel used by the network proxy
4. Initialization of the network proxy with the remote endpoint by either calling connect or listen
   (a) network proxy does a `channel_bind` to the local endpoint supplying its own bulk callback handlers
   (b) network proxy initializes the network part by starting the connection protocol or listen for connection requests
   (c) upon connection, the callback from the proxy is called signalling channel connection
5. the application supplies the network proxy with buffers by passing them on the local channel.

In contrast to normal channels, when using the proxy additional work has to be done: in case of the local channel the remote endpoints have to be created by the application as well. Further there are two additional functions which are specific to the backend that need to be invoked upon channel creation. These are shown in Listing 4.1. We have omitted the parameters in this listing, because of their number.

```
errval_t bulk_net_proxy_listen( ... );

errval_t bulk_net_proxy_connect( ... );
```

Listing 4.1: The Additional Proxy Functions

Usage

After the channels are setup and the proxies connected the channel is ready for use. From that point on the usage of the bulk channel does not differ much from any other channel type. However, the application needs to make sure that there are always enough bulk buffers at the network proxy in order to receive the arriving data.

Tradeoffs

There are several tradeoffs with this implementation:

- **Transparency:** This design goal is clearly not satisfied: the channel consists of three hops with two master endpoints.
- **Complicated Setup:** The channel setup needs additional function calls and differs from the common interface.

Notice, that even though the bulk channels designed to have as little overhead as possible, every additional hop adds some overhead to the operation.

4.4.2 Transparent Variant

The proxy variant as described in the previous section does not full-fill the channel semantics specified in the formal specification. We want to provide a fully transparent channel, which behaves semantically like it is machine local.

Overview

The main goal of this channel implementation is to provide the exact semantics a bulk channel should have: when a buffer with $id = 3$ is transmitted, then the other endpoint will receive the data in the buffer with $id = 3$. This implies that we need to establish a one-to-one relationship between the buffers and pools on one machine and the buffers and pools on the other machine[3]. An overview can be seen in Figure 4.3.

Channel Initialization

In contrast to the proxy variant, there is no special way to create the channel. During the channel establishment process the following steps are executed at the creating side and the binding side. Notice, that in addition to the points 1 and 2 which are also done in the proxy variant, the allocation of dedicated receive buffers is also necessary.

1. Initialization of the hardware queue and buffer descriptor rings
2. Installing a hardware filter which associates the port with the queue
3. Allocation of receive buffers of a pre defined size
4. Adding the receive buffers to the receive queues

Pool Assignment

When a new pool is assigned on the channel there are several steps that have to be done at the other side. The steps that need to be done are depending if the pool was never been assigned to a channel which has an endpoint in this domain. In any case, the pool is added to the pool list of the channel in the end if the application does not veto the assignment request. The following steps are executed if the pool was not present in this domain:

[3]Only for assigned pools over the network channel

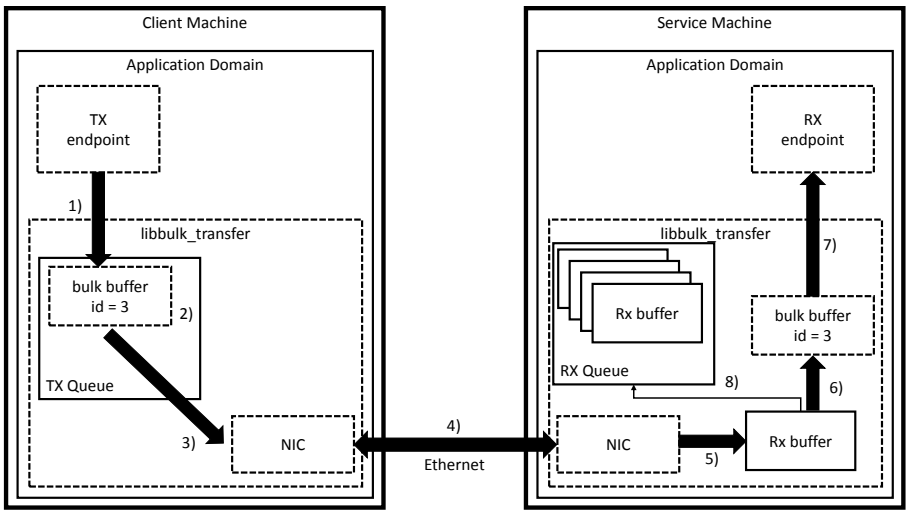

Figure 4.3: Transparent Network Channel

1. Allocate a virtual and physical address range.
2. Map the pool depending on the trust level.

It is important to know, that it is only allowed to assign pools with a bulk buffer size that is equal to the receive buffer size.

Sending a Buffer

The sending process is illustrated on the left hand side of Figure 4.3. When a buffer is moved or copied over the channel, it is handed over to the bulk transfer library (1). The network backend adds meta data and the header to the packet and enqueues the bulk buffer into the transmit queue (2). The NIC processes the transmit queue (3) and sends the packet onto the Ethernet (4).

Receiving of a Buffer

The receiving process can be seen on the right hand side of figure 4.3. When a new packet arrives on the network interface. The network hardware splits the header from the payload and stores the data into the next receive buffer[4] (5). The receive thread gets notified about the receive event and copies the data into the buffer with the identical id[5] of the one that was being sent (6). The callback to the application is executed (7) and the receive buffer added to the receive queue (8).

Tradeoffs

- **Resource Usage:** By strictly following the channel semantics each time a buffer gets transferred to the other machine, the allocated memory stays

[4]Notice there are in fact two needed: one for the buffer and one for the meta data.
[5]Same pool id and same offset

unavailable at the originating machine even though this memory is not used.

- **Receive Buffers:** Allocating receive buffers increases the resource consumption at both sides. Further, the number of allocated receive buffers must be chosen large enough to handle bursts.
- **Required Copy:** In the receive process, there is always a copy necessary.

Remarks about the Needed Copy

This variant violates the "zero copy" goal of a bulk transfer implementation. However, its easy to say that this bad. When we investigate possible use cases the answer is rather: it depends.

Assume, the data is received in a pass through domain and is never touched perhaps routed to another channel using the meta data. Because we do an actually copy the CPU is busy doing the copy which also pollutes the cache.

On the other hand, assume that the data is received in the sink domain. Because we are doing a copy, the data is expected to be already in the cache resulting in a lower cash miss rate when actually doing the data processing. Thus the copy can be viewed as a prefetcher.

4.4.3 No Copy Variant

The third variant we implemented tries to eliminate the need for copying from a receive buffer into the bulk buffer which was necessary in the transparent variant as described in the Section above while providing most of the bulk channel semantics.

Overview

To overcome the need for a copy, the data must be directly written to the bulk buffer by the network card. Therefore, the buffers must be enqueued into the receive queue before any data can be received. In contrast of the previous variants which make use of an in-line control channel, this variant consists of a distinct control channel. The receive buffers for this control channel are provided internally. In addition to that, for each pool a data channel where the bulk buffers serve as receive buffers. Figure 4.4 shows an overview of this variant from a data channel point of view.

Channel Initialization

In general there is almost no difference in initializing a channel to the procedure presented with the transparent variant. One thing, that one may optimize is the number and size of the internal receive buffers: the buffers must not have a size that is large enough to hold a bulk buffer but just large enough to hold the control messages and meta data.

Pool Assignment

As already explained in the overview, this variant makes use of distinct data channels. Therefore every time a new pool is assigned to this channel type

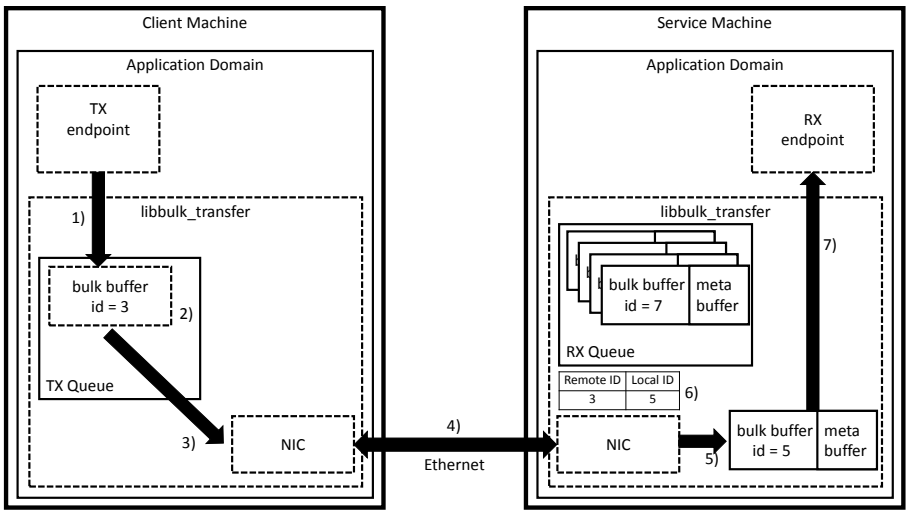

Figure 4.4: No Copy Variant (Pool View)

the a slightly different "channel setup" procedure needs to be executed. This establishes a port/queue - pool relationship: knowing the port/queue of a packet tells exactly which pool it belongs to and vice versa. The direction of the channel play an important role here, because if it is a transmit type channel the buffers must be added to the receive queue on the receive side.

Further, we need additional buffers to hold the meta data which also need to be allocated and added to the receive queue. There exists a one-to-one relation ship between bulk buffers and meta buffers i.e. foreach bulk buffer there is a meta buffer. The two buffers types, bulk and meta buffers, are added to the receive queue in alternating fashion with the bulk buffer first. The reason for this is can be seen when receiving a buffer (Section 4.4.3 below).

Sending Buffers

Sending a buffer is quite straight forward and basically behaves the same way as in the transparent variant. However, we must make sure that that we send the correct buffer id along, thus we need to look it up in the buffer id translation table[6] and send the remote id instead of the local one. This is especially important, when this buffers was received and passed back as a response.

Receiving Buffers

The receive process is illustrated on the right hand side of Figure 4.4. When a new packet arrives on the data channel (4) the header gets split away and the NIC takes the first buffer in the receive queue to fill the payload. (5) Because the bulk buffer is in the first part of the packet and a bulk buffer is always at first of the receive queue the bulk data gets written directly into the correct

[6]when the pool is assigned, on the transmission side the ids are initialized as remote id = local id.

destination. When the buffer is full, we are at the beginning of the meta data part, and the next receive buffer is dequeued (5) which is the internal meta buffer we allocated.

However, we need to provide correct semantics on a channel: when a applications sends buffer 3 over a channel and receives it back, it should also get the buffer 3 back: We need to overcome the fact that the receiving side does not know which buffer will arrive and just takes the next from the queue. Therefore we have to maintain a lookup table (6) which stores the *local* $\rightarrow$ *remote* id relation ship. After this book keeping is done, the application gets informed about the received data (7).

Potential Benefits

As described above, this variant makes use of a new receive queue per added pool and an additional one for the control channel. This setup of having multiple receive queues may be beneficial for the performance. It essentially enables concurrent processing in contrast to the other variants where all packets arrive over the same queue.

Problematic Aspects

There are several issues with that approach. First, the library must know exactly the location of all the buffers belonging to this pool at assign time: The buffers cannot be added to the receive queue if they are used anywhere else in the system. If the pool is added to a transmit channel, then the pool can only be added, if it is not already present in the receiving domain. Otherwise its not sure which buffers are available and can be added to the receive queue. No buffers in the queue implies no possibility to receive data.

Secondly, the way the receive queue is set up heavily relies on the packet layout of arriving data. If there is at some point of time less data arrived than the size of the bulk buffer, the queue gets out of sync resulting in a corrupted channel. This can also occur when a malicious machine injects a packet into the data stream.

Tradeoffs

Avoiding the copying brings certain tradeoffs:

1. **Pool Assignments:** Not all pools can be assigned to this channel: only the ones that are first assigned to the network channel.
2. **Packet Format:** The packets on the data channels must have always exactly the same format: [Buffer | Meta Data].
3. **Resource Usage:** There are also additional buffers needed to hold the meta data and further there is a queue allocated for each pool.

Chapter 5

Evaluation

In this Chapter we present the results from our evaluation. We conducted experiments for each of the implementation variants described in Chapter 4. As performance metrics we have chosen throughput (channel bandwidth) and the round trip time (response time).

5.1 Experiments

5.1.1 Experiment Factors

We have chosen the following factors for the experiment (Table 5.1). As baseline comparison we have taken the shared memory backend.

Factor	Values
Backend	Proxy, Transparent, No-Copy
Buffer Size	4kB, 8kB

Table 5.1: Experiment Factors

5.1.2 Genera Experiment Setup

There are two domains involved in this experiment. First we have a server, the "ECHO" domain which basically replies the request i.e. sending the buffer back. Secondly there is the test domain, which runs the experiment and measures the times. There is a single integer value written to the buffer and the value is read on the receive event. A checksum over these values ensures the correctness.

5.1.3 Throughput

The first response variable we want to estimate is the throughput of the different backends which we may call channel bandwidth. Since we are using a 10Gb NIC this will be the upper limit of achievable performance.

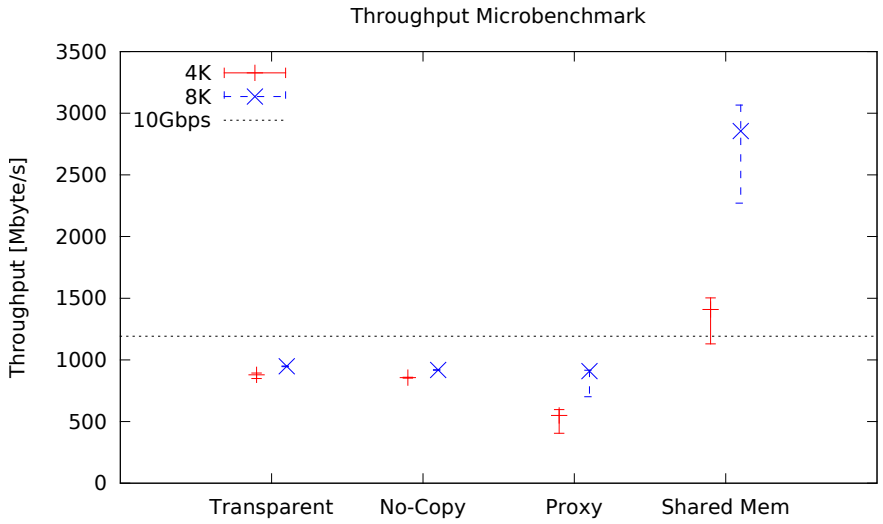

Figure 5.1: Channel Throughput

Experiment Setup

The experiment was conducted with a repetition of 1000 runs. Each run consists of 250 overlapping move operations in both ways. The time was measured from the point when the first buffer was sent up to receiving back the last buffer. The trust level of the channel was set to fully trusted. The shared memory channel as a comparison is run between two different cores.

Results

The measurements of the experiment can be seen in Figure 5.1. The graph shows the theoretical maximum of the 10Gbps link shows as the dashed horizontal line. The data points are the medians while the error bars show the 95 percentiles. One may notice that the measurements are very stable.

We see a slight difference between the 4kB and 8kB buffer size. The throughput values in the graph are the values of usable data i.e. only the buffer data is included in the calculations. We see that there is a slight increase in the throughput when we increase the buffer size. This results in a slight lower overhead introduced by the headers and trailers per buffer byte transmitted. Overall we can say that we got close to the maximum available performance.

As an interesting coincidence we see that the throughput values of the 8kB buffers is almost the same with every implementation. While the proxy backend has a significantly lower throughput in the 4kB case than the other backends. We account this to the fact that the received buffers need to be transferred the local twice.

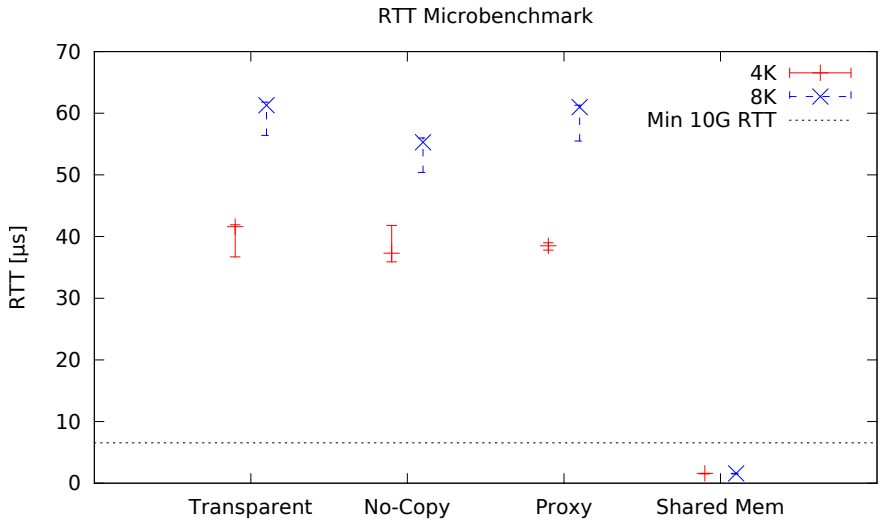

Figure 5.2: Channel Round Trip Time

5.1.4 Round Trip Time

The second response variable is the roundtrip time (RTT). Notice, that we are more interested in the delay i.e. the time it takes to move a buffer from one machine to another. We can estimate the delay by dividing the measured RTT by two.

Experiment Setup

As with the previous experiment, the RTT experiment was conducted with a repetition of 1000 runs while each run consists of the measurement of 250 move operations. In contrast to the throughput, we measure the time it takes to get the reply back and do not issue overlapping operations.

Results

The experiment results can be seen in Figure 5.2. First of all we see that the shared memory clearly outperforms the network implementation which is obvious since there is a significant delay due to the transmission time. The dashed line shows the base line when sending 4kB over a 10Gbps link without any overhead.

With the 4kB buffers, we see a very consistent picture. All the three backend variants perform not significantly different. With 8kB buffer size, there is a difference, the no-copy variant performs slightly better. 95% of the measurements in the no copy case are blow the fastest 5% of the other two implementations. Recall, that there is an additional queue per assigned pool. This enables some concurrency which may lead to the slightly better RTT.

Further the difference between two buffer sizes is not a factor of two. This indicates that quite some time is spent processing the packet. Under the as-

sumption that the processing overhead stays the same with a bigger buffer size, we get a processing time of about 15-20μs per request.

5.1.5 Block Service

In addition to the two micro benchmarks above, we also ran an experiment on our block service implementation. In contrast to the description of the block service in the common part, we just make use of the NBS client domain that issues the benchmark requests, because we want examine the the performance of the network channel.

Experiment Setup

Again the experiment is repeated for 1000 runs. Each run consists of 250 write or read requests respectively. A write request will allocate a new bulk buffer, fill the entire buffer with sample data based on the block id we want to write and then move it over the block server. The read requests are executed as a batch request of 250 blocks. The contents of the received payloads are examined and compared to the expected value which was written by the write request to ensure correctness.

Results

The result of this benchmark are shown in Figure 5.3. Interestingly we come almost close to the theoretic maximum for the write requests. We measured from the bulk move request up until we got the pass back of the buffer. In the proxy case this measures the loop back of the local channel i.e. the time it takes till the packet is sent which is obviously at 10Gbps. While the transparent channel provides consistent read and write performance as expected, the no copy variant delivers has about twice as high write performance than read performance, which seems a bit suspicious to us. We think that with that benchmark was something wrong in the measurement process.

Further, since the read request is sent over lwIP TCP connection this may have an influence on the read performance. Because the TCP/IP processing usually takes more time.

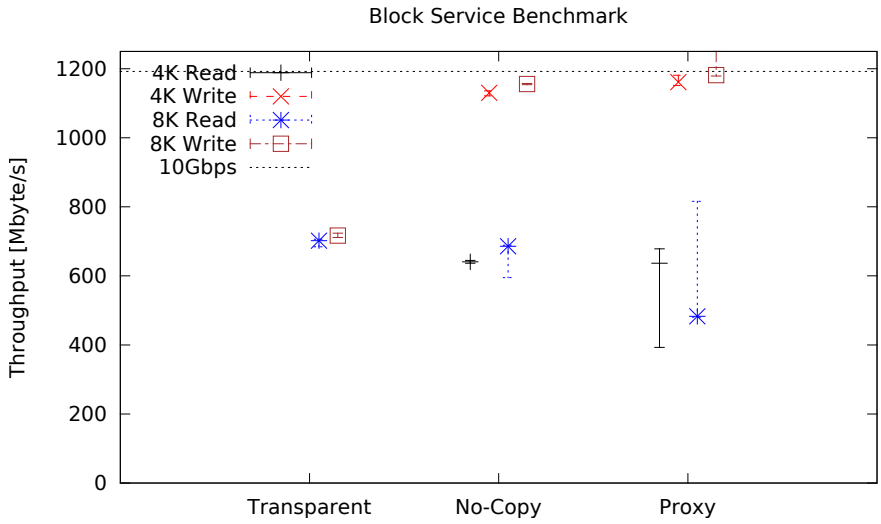

Figure 5.3: Block Service Performance

Chapter 6

Future Work

6.1 Receive Buffers for Meta Data

If a buffer is transmitted, the received packed always ends up in two receive buffers: one for the bulk buffer and the other for the meta data. What we not want is that a packet that arrives, occupies two bulk buffers at the receiving side (even if temporary). Under the assumption that all pools assigned to a network channel have equal buffer sizes, for every packet that arrives we know exactly where the meta data starts. Therefore having a hardware feature that does not only a header split but also a custom packet split would ensure that only bulk buffer data part will end up in a bulk buffer. The buffer chaining with FIFO characteristics of the receive queue may end up in something that is similar to the desired feature, however relies that the arriving packets have a specific format and there is no interference with other packets that do not match this format (e.g. control packets vs. data packets).

6.2 Maximum Buffer Size

Recall Section 3.1.2 about hardware support for jumbo frames and the buffer size constraints. This two constraints give us a a support only two sizes: 4096 and 8192 bytes for the non-trusted case. Note that the hardware supports a maximum Ethernet frame size of just below 16kB.

If we just consider the trusted case, we may relax the page size constraint a bit and allow buffer sizes which are not a multiple of page size. However, by relaxing this constraints, we give away the possibility to enforce protection by doing page table manipulations.

6.3 Security Aspects: Use of IO-MMU

We want to give guarantees that at the point a bulk buffer is transferred to another domain - either copy or move - its contents are not modified by the sending domain. While this can be enforced by setting the access rights on a software level, a hardware device does just know about accessing physical memory directly. Therefore by supplying the hardware with the "wrong" physical

address can lead to data corruption.

Thus in the non trusted case the use of a IO-MMU should be encouraged to protect the memory region from malicious DMA access. It may be worth investigating the influence of performance with changing the IO page tables in addition to changing the domain page tables. Conceptually, you will need to adapt the IO page tables whenever a buffer is handed over to the network backend or arrives on the network backend.

6.4 The No-Copy Variant Issues

There are several limitations with the no copy variant. Recall, the restriction that a pool can only be assigned to such a channel at maximum once and only if the pool has not yet been assigned to the domain. This is a hard limitation. Further that channel does not support any operation that involves removing a buffer out of the receive queue such as passing a buffer over a transmit channel.

6.5 RDMA Backend

Our implementation using UDP involves a bit of work by the two host machines i.e. queue manipulations, packet processing and so forth. Accessing the buffer directly in the main memory of the remote host would be a great way to implement a bulk transfer mechanism across machines. By the use of RDMA, the contents of a buffer can be precisely loaded into the exact same buffer as on the other host, which was not the case in our implementations (Sections 4.4.3 and 4.4.2) where we either used buffer id rewriting or a copy. In addition to that, the meta data can be placed into a designated memory area.

6.6 Extended Network Support

The current support of buffer sizes is rather small and limited (Section 6.2). To overcome the limitations of 4096 or 8192 byte buffers, the implementation of fragmentation handling would be a possible step. Having fragmentation support not only avoids the use of jumbo frames, but also enables bigger buffer sizes. On the other hand, as explained the use of jumbo frames can lead to a better performance and the effort dealing with the additional complexity ma not be worth it.

6.7 Network Stack

Recall, the current network stack makes use of pbufs to receive data and pass it to the application. As already discussed, the pbufs have some limitations [22]. It may be worth investigating if the network stack could be adapted to deal with those bulk buffers and what this would mean to performance, protocol processing and data security.

6.8 Freeing Up Resources

The current implementation of the network backends do not support the deallocation of resources such as hardware queues or port. Further the backend does not support the messages for channel teardown and pool removal. This is to be considered a valid point for future work, to give a full featured implementation with the possibility to free up resources.

Bibliography

[1] Mark Nevill Akhilesh Singhania, Ihor Kuz. *Capability Management in Barrelfish, Barrelfish Technical Note 013.* Barrelfish Project, ETH Zurich, Systems Group, Department of Computer Science, Universitätsstr 6, 8092 Zurich, 12 2013.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[3] Andrew Baumann. *Inter-dispatcher communication in Barrelfish, Barrelfish Technical Note 011.* Barrelfish Project, ETH Zurich, Systems Group, Department of Computer Science, Universitätsstr 6, 8092 Zurich, 12 2011.

[4] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn't your os? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verità, Switzerland, 04 2009.

[5] Motti Beck and Michael Kagan. Performance evaluation of the RDMA over Ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, DC-CaVES '11, pages 9–15. International Teletraffic Congress, 2011.

[6] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol implementation in a vertically structured operating system. In *In Proc. 22nd Annual Conference on Local Computer Networks*, pages 179–188, 1997.

[7] Ron Brightwell, Bill Lawry, Arthur B. MacCabe, and Rolf Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 268–, Washington, DC, USA, 2002. IEEE Computer Society.

[8] Intel Corp. Intel® 82599 10 gigabit ethernet controller family. `http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gigabit-ethernet-controller-family.html`, February 2014.

[9] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 189–202, New York, NY, USA, 1993. ACM.

[10] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance characterization of a 10-gigabit Ethernet TOE. In *Proceedings of the 13th Symposium on High Performance Interconnects*, HOTI '05, pages 58–63, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, PPAM'05, pages 228–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[12] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, Berkeley, CA, USA, 2012. USENIX Association.

[13] Tim Harris, Martín Abadi, Rebecca Isaacs, and Ross Mcilroy. Ac: Composable asynchronous io for native languages.

[14] Jonas Hauenstein, David Gerhard, and Gerd Zellweger. Ethernet message passing for Barrelfish. Distributed systems lab, ETH Zurich, July 2011.

[15] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network I/O. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 50–59, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Antoine Kaufmann. Low-latency OS protocol stack analysis. Bachelor thesis, ETH Zurich, January 2012.

[17] Oracle. Java byte buffers. `http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html`.

[18] Ian Pratt and Keir Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 67–76. IEEE, 2001.

[19] The FreeBSD Project. mbuf – memory management in the kernel ipc subsystem. `http://www.freebsd.org/cgi/man.cgi?query=mbuf&sektion=9&manpath=FreeBSD+10.0-RELEASE`.

[20] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[21] Andrew Baumann; Paul Barham; Pierre-Evariste Dagand; Tim Harris; Rebecca Isaacs; Simon Peter; Timothy Roscoe; Adrian Schüpbach and Akhilesh Singhania. The multikernel: A new os architecture for scalable

multicore systems. In *Proceedings of the 22nd ACM Symposium on OS Principles*, Big Sky, MT, USA, 10 2009.

[22] Pravin Shinde. *Bulk Transfer, Barrelfish Technical Note 014*. Barrelfish Project, ETH Zurich, Systems Group, Department of Computer Science, Universitätsstr 6, 8092 Zurich, 08 2011.

[23] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We need to talk about NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 1–1, Berkeley, CA, USA, 2013. USENIX Association.

[24] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. ACM.