Technical Report
Distributed Systems Lab

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Achermann Reto, Kaufmann Antoine

Bulk Transfer over Shared Memory

by

Bär Jeremia, Föllmi Claudio

Supervised by

Prof. Timothy Roscoe, Dr. Kornilios Kourtis

February 14, 2014

inf | Informatik
Computer Science

Contents

I Bulk Transfer Specification **6**

1 Introduction **12**

2 Related Work **13**

3 Barrelfish Operating System **15**
- 3.1 Operating System Structure . . . 15
- 3.2 Flounder . . . 15
- 3.3 Resource Management and Capabilities . . . 16
- 3.4 Current Bulk Transfer Implementation . . . 16
  - 3.4.1 The Official Bulk Transfer Infrastructure . . . 16
  - 3.4.2 Networking Bulk Transfer: pbufs . . . 17
  - 3.4.3 Summary . . . 17

4 Architectural Design **18**
- 4.1 Design Goals . . . 18
- 4.2 Terminology . . . 18
  - 4.2.1 Backend . . . 18
  - 4.2.2 Endpoint . . . 19
  - 4.2.3 Channel . . . 19
  - 4.2.4 Channel Properties . . . 20
  - 4.2.5 Pool . . . 21
  - 4.2.6 Buffer . . . 21
- 4.3 Bulk Transfer Architecture . . . 21
- 4.4 Additional Libraries . . . 22

5 Semantic Specification **24**
- 5.1 Formal Specification . . . 24
  - 5.1.1 Buffer States . . . 24
  - 5.1.2 Formal State Definition . . . 25
  - 5.1.3 Invariants . . . 25
  - 5.1.4 Channel Creation and Binding . . . 27
  - 5.1.5 Pool Allocation and Assignment . . . 27
  - 5.1.6 Buffer Transfer Types . . . 28
  - 5.1.7 Moving a Buffer on a Channel . . . 28
  - 5.1.8 Buffer Pass . . . 29
  - 5.1.9 Read-Only Copy . . . 29
  - 5.1.10 Copy Release . . . 31

5.1.11 Full Copy 31
5.2 Overview of Operations 31
5.2.1 Buffer State Diagram 31
5.2.2 Channel Setup 31
5.2.3 Pool Assignment 33
5.2.4 Sending and Receiving 34

6 Interface Specification 37
6.1 Core Functionality 37
6.2 Core Data Structure 37
6.2.1 Enumerations and Constants 38
6.2.2 Bulk Channel 39
6.2.3 Bulk Endpoint Descriptors 40
6.2.4 Bulk Pool and Bulk Buffer 40
6.3 Bulk Channel Operations 42
6.3.1 Asynchronous Interfaces 42
6.3.2 Channel Initialization and Teardown 43
6.3.3 Pool Assignment and Removal 43
6.3.4 Buffer Transfer Operations 44
6.4 Pool Allocator 44
6.4.1 Allocator Initialization 45
6.4.2 Allocating and Returning Buffers 45
6.5 The Bulk Transfer Library 45
6.5.1 Contracts of Public Interface Functions 45
6.5.2 Buffer Related Functions 46
6.5.3 Pool Related Functions 46

7 Application 48
7.1 Domains 48
7.2 Connection Initialization 49
7.3 Pool Assignments 50
7.4 Basic Work Load 50

8 Conclusion 51
8.1 Support for Variable Sized Buffers 51
8.2 Enforcing Policies 51
8.3 Device Drivers 51
8.4 Aggregate Objects 52
8.5 Name Service Integration 52
8.6 THC Integration 52

II Shared Memory Bulk Transfer 53

1 Introduction 57

2 Related Work 58
2.1 fbufs 58
2.2 rbufs 58

3 Design Considerations **60**
3.1 General Design Challenges 60
3.2 Capability System 60
3.3 Using Capabilities For Bulk Buffers 61
3.4 Trusted Third Party 61
3.5 New Capability 62
3.6 Limitations 64

4 Implementation **65**
4.1 Existing Infrastructure 65
4.1.1 Libbarrelfish Bulk Transfer 65
4.1.2 Flounder 65
4.2 Backend Implementation 66
4.2.1 Overall Design 66
4.2.2 Channel Creation 66
4.2.3 Transferring Buffers 67
4.2.4 Implementation Challenge: Message Ordering 68
4.2.5 Implementation Challenge 2: Unnecessary Locking Of Monitor 70

5 Evaluation **72**
5.1 Bulk Transfer Semantics 72
5.2 Bulk Transfer Performance 74
5.2.1 Setup 74
5.2.2 Testing Hardware 75
5.2.3 Results 77
5.2.4 Implications For Users 82
5.2.5 Possible Improvements 82

6 Future Work **83**

List of Figures

4.1 Bulk Transfer Architecture 20
4.2 Architecture Layers 22

5.1 Buffer State Diagram (Domain View) 32
5.2 Setup of a Bulk Channel 33
5.3 Pool Assign Operation 35
5.4 Bulk Transfer Architecture 36

6.1 Pool and Buffer Representation 41

7.1 Network Proxy using Local Channel 49

3.1 State diagram of a single shared_frame capability. 63

4.1 Messages transmitted between channel endpoints in a copy operation. 68

5.1 Interaction of the two agents in the `bulk_shm` test. 73
5.2 Microbenchmark configuration with communicating domains on the same core. 76
5.3 Microbenchmark configuration with communicating domains on different cores. 76
5.4 Microbenchmark configuration with additional hops between communicating domains, all on different cores. 76
5.5 Latency as a function of bulk buffer size. 78
5.6 Latency for different core configurations. 79
5.7 Throughput for different core configurations. 79
5.8 Latency for different core configurations. 81
5.9 Throughput for different core configurations. 81

List of Tables

5.1 The used formal specification 24

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

1 Introduction **12**

2 Related Work **13**

3 Barrelfish Operating System **15**

3.1 Operating System Structure 15

3.2 Flounder 15

3.3 Resource Management and Capabilities 16

3.4 Current Bulk Transfer Implementation 16

3.4.1 The Official Bulk Transfer Infrastructure 16

3.4.2 Networking Bulk Transfer: pbufs 17

3.4.3 Summary 17

4 Architectural Design **18**

4.1 Design Goals 18

4.2 Terminology 18

4.2.1 Backend 18

4.2.2 Endpoint 19

4.2.3 Channel 19

4.2.4 Channel Properties 20

4.2.5 Pool 21

4.2.6 Buffer 21

4.3 Bulk Transfer Architecture 21

4.4 Additional Libraries 22

5 Semantic Specification 24
5.1 Formal Specification . . . 24
5.1.1 Buffer States . . . 24
5.1.2 Formal State Definition . . . 25
5.1.3 Invariants . . . 25
5.1.4 Channel Creation and Binding . . . 27
5.1.5 Pool Allocation and Assignment . . . 27
5.1.6 Buffer Transfer Types . . . 28
5.1.7 Moving a Buffer on a Channel . . . 28
5.1.8 Buffer Pass . . . 29
5.1.9 Read-Only Copy . . . 29
5.1.10 Copy Release . . . 31
5.1.11 Full Copy . . . 31
5.2 Overview of Operations . . . 31
5.2.1 Buffer State Diagram . . . 31
5.2.2 Channel Setup . . . 31
5.2.3 Pool Assignment . . . 33
5.2.4 Sending and Receiving . . . 34

6 Interface Specification 37
6.1 Core Functionality . . . 37
6.2 Core Data Structure . . . 37
6.2.1 Enumerations and Constants . . . 38
6.2.2 Bulk Channel . . . 39
6.2.3 Bulk Endpoint Descriptors . . . 40
6.2.4 Bulk Pool and Bulk Buffer . . . 40
6.3 Bulk Channel Operations . . . 42
6.3.1 Asynchronous Interfaces . . . 42
6.3.2 Channel Initialization and Teardown . . . 43
6.3.3 Pool Assignment and Removal . . . 43
6.3.4 Buffer Transfer Operations . . . 44
6.4 Pool Allocator . . . 44
6.4.1 Allocator Initialization . . . 45
6.4.2 Allocating and Returning Buffers . . . 45
6.5 The Bulk Transfer Library . . . 45
6.5.1 Contracts of Public Interface Functions . . . 45
6.5.2 Buffer Related Functions . . . 46
6.5.3 Pool Related Functions . . . 46

7 Application **48**
7.1 Domains . . . 48
7.2 Connection Initialization . . . 49
7.3 Pool Assignments . . . 50
7.4 Basic Work Load . . . 50

8 Conclusion **51**
8.1 Support for Variable Sized Buffers . . . 51
8.2 Enforcing Policies . . . 51
8.3 Device Drivers . . . 51
8.4 Aggregate Objects . . . 52
8.5 Name Service Integration . . . 52
8.6 THC Integration . . . 52

List of Figures

4.1 Bulk Transfer Architecture 20
4.2 Architecture Layers 22
5.1 Buffer State Diagram (Domain View) 32
5.2 Setup of a Bulk Channel 33
5.3 Pool Assign Operation 35
5.4 Bulk Transfer Architecture 36
6.1 Pool and Buffer Representation 41
7.1 Network Proxy using Local Channel 49

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* [6] 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 [5] 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 [9] 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 [10]. 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 to it and send messages, either in an asynchronous fashion or with full RPC semantics.

[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/`

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 [11] 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 [11] 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

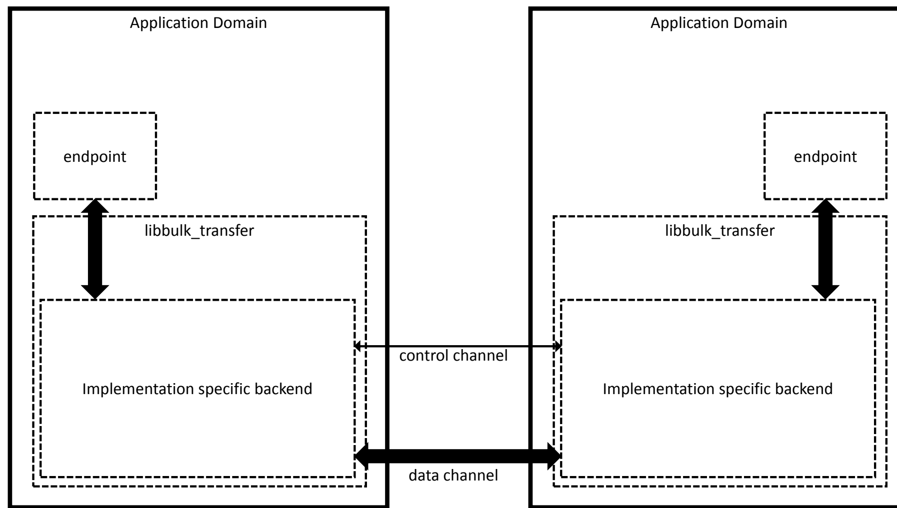

Figure 4.1: Bulk Transfer Architecture

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.

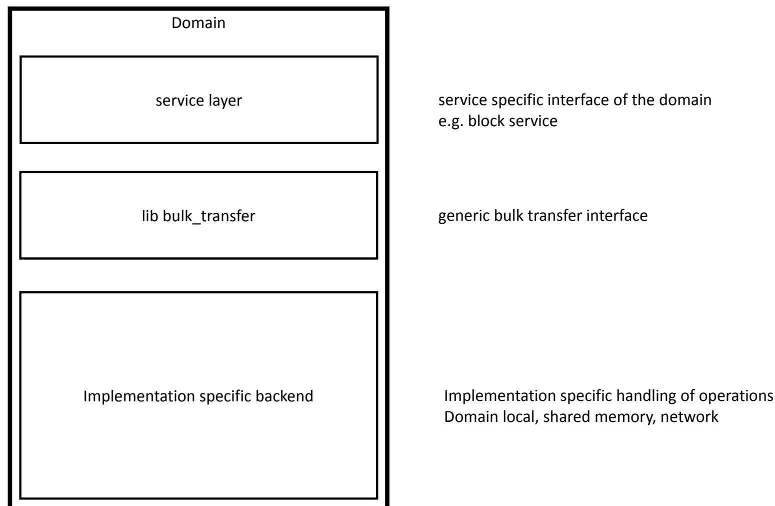

Figure 4.2: Architecture Layers

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 [6]. 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.

```
struct bulk_aggregate_object
{
  size_t num_bufs;
  struct
  {
    struct bulk_pool_id pool_id;
    size_t              buffer_id;
  }      buffers[];
}
```

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: $\mathrm{ACCESS_RIGHTS} := \{\mathrm{NONE}, \mathrm{READ}, \mathrm{READ_WRITE}\}$
2: $\mathrm{ROLES} := \{\mathrm{MASTER}, \mathrm{SLAVE}\}$
3: $\mathrm{DIRECTIONS} := \{\mathrm{SOURCE}, \mathrm{SINK}\}$
4:
5: $\mathrm{valid_buffers} \subseteq \mathrm{BUFFERS}$
6: $\mathrm{valid_pools} \subseteq \mathrm{POOLS}$
7: $\mathrm{valid_channels} \subseteq \mathrm{CHANNELS}$
8: $\mathrm{valid_eps} \subseteq \mathrm{ENDPOINTS}$
9:
10: $\mathrm{buffers_to_pools} \in \mathrm{valid_buffers} \rightarrow \mathrm{valid_pools}$
11: $\mathrm{buffer_ownership} \in \mathrm{valid_buffers} \rightarrow \mathrm{DOMAINS}$
12: $\mathrm{buffer_access} \in (\mathrm{valid_buffers} \times \mathrm{DOMAINS}) \rightarrow \mathrm{ACCESS_RIGHTS}$
13: $\mathrm{buffer_data} \in \mathrm{valid_buffers} \rightarrow \mathrm{DATA}$
14: $\mathrm{buffer_copies} \subseteq \mathrm{valid_buffers} \times \mathrm{valid_channels}$
15: $\mathrm{pools_to_channels} \subseteq \mathrm{valid_pools} \times \mathrm{valid_channels}$
16: $\mathrm{ep_to_dom} \in \mathrm{valid_ep} \rightarrow \mathrm{DOMAINS}$
17: $\mathrm{dir_to_ep} \in (\mathrm{valid_channels} \times \mathrm{DIRECTIONS}) \nrightarrow \mathrm{valid_ep}$
18: $\mathrm{roles_to_ep} \in (\mathrm{valid_channels} \times \mathrm{ROLES}) \nrightarrow \mathrm{valid_ep}$
19: $\mathrm{data_queues} \in \mathrm{valid_channels} \rightarrow \mathrm{BUFFER_QUEUES}$
20: $\mathrm{control_queues} \in \mathrm{valid_channels} \rightarrow \mathrm{BUFFER_QUEUES}$

Algorithm 2 Invariants

1: $\mathrm{copy_receivers}(\mathrm{b}, \mathrm{buffer_copies}') :=$
$\mathrm{ep_to_dom}[\mathrm{dir_to_ep}[\mathrm{buffer_copies}'[\{\mathrm{b}\}] \times \{\mathrm{SINK}\}]]$
2: $\forall\, \mathrm{b}, \mathrm{d} : \mathrm{buffer_ownership}(\mathrm{b}) = \mathrm{d} \wedge \mathrm{b} \notin \mathrm{dom}(\mathrm{buffer_copies}) \Leftrightarrow$
$\mathrm{buffer_access}(\mathrm{b}, \mathrm{d}) = \mathrm{READ_WRITE}$
3: $\forall\, \mathrm{b}, \mathrm{d} : (\mathrm{buffer_ownership}(\mathrm{b}) = \mathrm{d} \wedge \mathrm{b} \in \mathrm{dom}(\mathrm{buffer_copies})) \vee$
$\mathrm{d} \in \mathrm{copy_receivers}(\mathrm{b}, \mathrm{buffer_copies}) \Leftrightarrow \mathrm{buffer_access}(\mathrm{b}, \mathrm{d}) = \mathrm{READ}$
4: $\forall\, \mathrm{b}, \mathrm{d} : \mathrm{buffer_ownership}(\mathrm{b}) \neq \mathrm{d} \wedge \mathrm{d} \notin \mathrm{copy_receivers}(\mathrm{b}, \mathrm{buffer_copies}) \Leftrightarrow$
$\mathrm{buffer_access}(\mathrm{b}, \mathrm{d}) = \mathrm{NONE}$
5: $\mathrm{copy_reachability_satisfied}(\mathrm{buffer_copies}') :=$
$\forall\, \mathrm{b}, \mathrm{c} : (\mathrm{b}, \mathrm{c}) \in \mathrm{buffer_copies}' \Rightarrow$
$\exists\, \mathrm{d} : \mathrm{d} = \mathrm{ep_to_dom}(\mathrm{dir_to_ep}(\mathrm{c}, \mathrm{SOURCE})) \wedge$
$(\mathrm{buffer_ownership}(\mathrm{b}) = \mathrm{d} \vee \mathrm{d} \in \mathrm{copy_receivers}(\mathrm{b}, \mathrm{buffer_copies}'))$

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

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 $\in$ DOMAINS
Require: local_ep $\in$ (ENDPOINTS $\setminus$ valid_eps)
Require: remote_ep $\in$ valid_eps
Require: $\exists\,\mathrm{chan}, \mathrm{remote_dir}, \mathrm{remote_role}:$
Require: $\mathrm{dir_to_eps}(\mathrm{chan}, \mathrm{remote_dir}) = \mathrm{remote_ep}$
Require: $\mathrm{roles_to_eps}(\mathrm{chan}, \mathrm{remote_role}) = \mathrm{remote_ep}$
Require: $(\mathrm{chan}, \mathrm{other_dir}(\mathrm{remote_dir})) \notin \mathrm{dom}(\mathrm{dir_to_eps})$
Require: $(\mathrm{chan}, \mathrm{other_role}(\mathrm{remote_role})) \notin \mathrm{dom}(\mathrm{roles_to_eps})$
1: **procedure** ChannelBind(domain, local_ep, remote_ep)
2: **end procedure**
Ensure: local_ep $\in$ valid_eps’
Ensure: $\mathrm{dir_to_eps'}(\mathrm{chan}, \mathrm{other_dir}(\mathrm{remote_dir})) = \mathrm{local_ep}$
Ensure: $\mathrm{dir_to_eps'}(\mathrm{chan}, \mathrm{remote_dir}) = \mathrm{remote_ep}$
Ensure: $\mathrm{roles_to_eps'}(\mathrm{chan}, \mathrm{other_role}(\mathrm{remote_role})) = \mathrm{local_ep}$
Ensure: $\mathrm{roles_to_eps'}(\mathrm{chan}, \mathrm{remote_role}) = \mathrm{remote_ep}$
Ensure: $\mathrm{eps_to_domains}'(\mathrm{local_ep}) = \mathrm{domain}$

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 $\in$ DOMAINS
Require: pool $\in$ (POOLS $\setminus$ valid_pools)
Require: bufs $\subseteq$ (BUFFERS $\setminus$ valid_buffers) $\wedge$ bufs $\neq \varnothing$
1: **procedure** PoolCreate(domain, pool, bufs)
2: **end procedure**
Ensure: pool $\in$ valid_pools$'$
Ensure: bufs $\subseteq$ valid_buffers$'$
Ensure: $\forall\,\mathrm{b} : \mathrm{b} \in \mathrm{bufs} \Rightarrow \mathrm{buffer_to_pools}(\mathrm{b}) = \mathrm{pool}$
Ensure: $\forall\,\mathrm{b} : \mathrm{b} \in \mathrm{bufs} \Rightarrow \mathrm{buffer_ownership}(\mathrm{b}) = \mathrm{domain}$

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: $\text{ep} \in \text{valid_eps}$
Require: $\text{pool} \in \text{valid_pools}$
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: $\text{ep} \in \text{valid_ep}$
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: $\text{ep} \in \text{valid_ep}$
Require: $\exists\, \text{domain} : \text{ep_to_dom}(\text{ep}) = \text{domain}$
Require: $\text{buffer_ownership}(\text{buffer}) = \text{domain}$
Require: $\exists\, \text{chan} : \text{dir_to_ep}^{-1}(\text{ep}) = (\text{chan}, \text{SINK})$
Require: $\exists\, \text{other_ep} : \text{dir_to_ep}(\text{chan}, \text{SOURCE})) = \text{other_ep}$
Require: $(\text{buffer}, \text{chan}) \in \text{buffers_to_pools} \circ \text{pools_to_channels}$
1: **procedure** ChannelPassBuffer(ep, buffer)
2: **end procedure**
Ensure: $\text{control_queues}'(\text{chan}) = \text{enqueue}(\text{control_queues}(\text{chan}), \text{buffer})$
Ensure: $\text{buffer_ownership}'(\text{buffer}) = \text{ep_to_dom}(\text{other_ep})$

Algorithm 9 Copy Buffer on Channel

Require: $\text{ep} \in \text{valid_ep}$
Require: $\exists\, \text{domain} : \text{ep_to_dom}(\text{ep}) = \text{domain}$
Require: $\text{buffer_ownership}(\text{buffer}) = \text{domain} \lor$
$(\exists\, \text{c} : (\text{buffer}, \text{c}) \in \text{buffer_copies} \land$
$\text{ep_to_dom}(\text{dir_to_ep}(\text{c}, \text{SINK})) =$
$\text{ep_to_dom}(\text{ep}))$
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** ChannelCopyBuffer(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: $\text{buffer_ownership}'(\text{buffer}) = \text{buffer_ownership}(\text{buffer})$
Ensure: $\text{buf} \neq \text{buffer} \Rightarrow \text{buffer_ownership}'(\text{buf}) = \text{ep_to_dom}(\text{other_ep})$
Ensure: $\text{buf} = \text{buffer} \Rightarrow$
$\text{buffer_copies}'[\{\text{buffer}\}] = \text{buffer_copies}[\{\text{buffer}\}] \cup \{\text{chan}\}$

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: $\mathrm{ep} \in \mathrm{valid_ep}$
Require: $\exists\, \mathrm{chan} : \mathrm{dir_to_ep}^{-1}(\mathrm{ep}) = (\mathrm{chan}, \mathrm{SINK})$
Require: $\exists\, \mathrm{other_ep} : \mathrm{dir_to_ep}(\mathrm{chan}, \mathrm{SOURCE})) = \mathrm{other_ep}$
Require: $(\mathrm{buffer}, \mathrm{chan}) \in \mathrm{buffer_copies}$
Require: $\mathrm{copy_reachability_satisfied}(\mathrm{buffer_copies} \setminus \{(\mathrm{buffer}, \mathrm{chan})\})$
Require: $(\mathrm{buffer}, \mathrm{chan}) \in \mathrm{buffers_to_pools} \circ \mathrm{pools_to_channels}$
1: **procedure** ChannelReleaseCopy(ep, buffer)
2: **end procedure**
Ensure: $(\mathrm{buffer}, \mathrm{chan}) \notin \mathrm{buffer_copies}'$
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

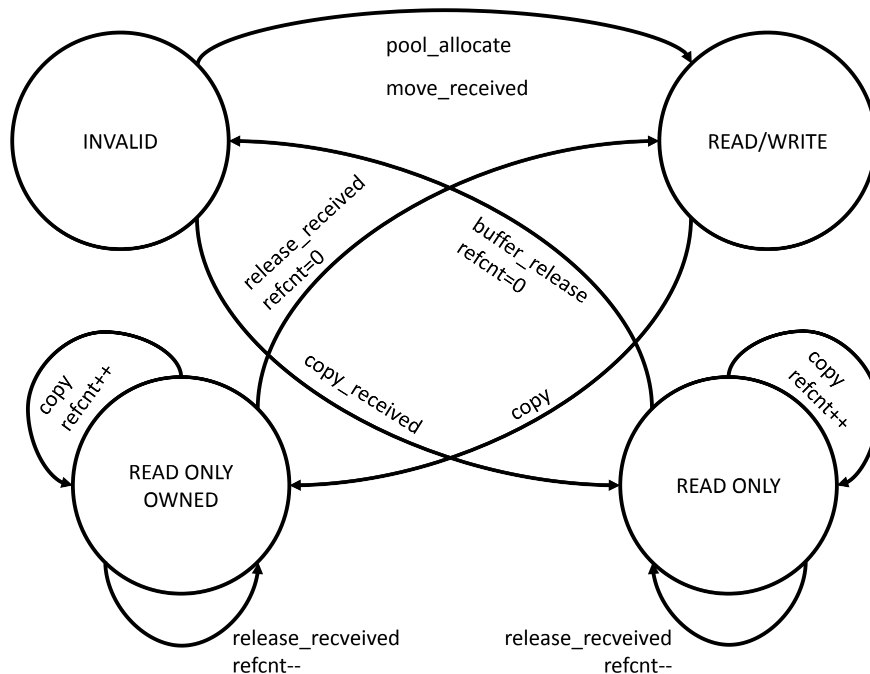

Figure 5.1: Buffer State Diagram (Domain View)

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.

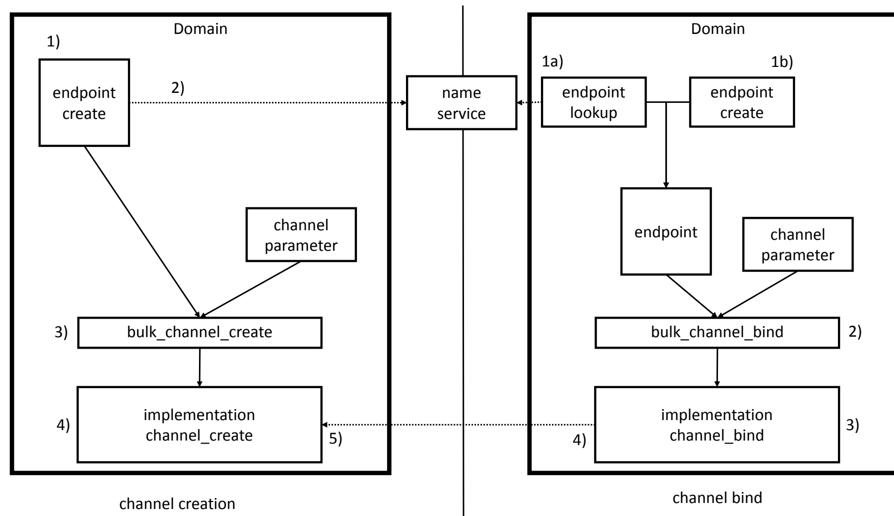

Figure 5.2: Setup of a Bulk Channel

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

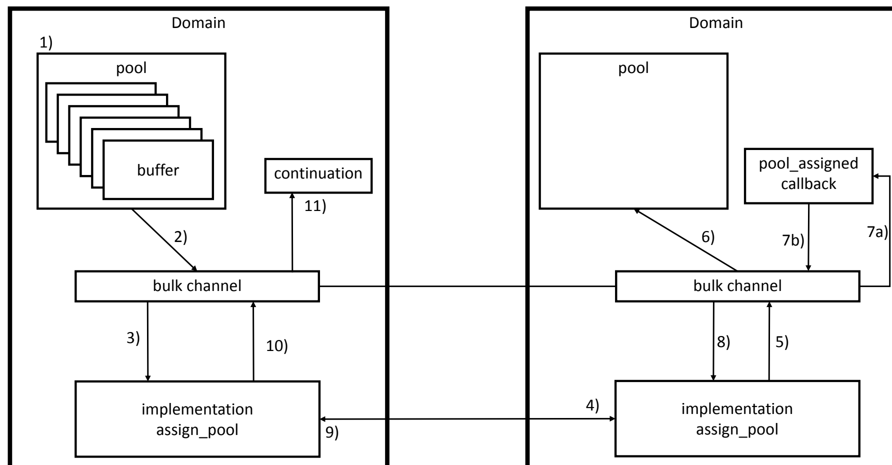

Figure 5.3: Pool Assign Operation

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.

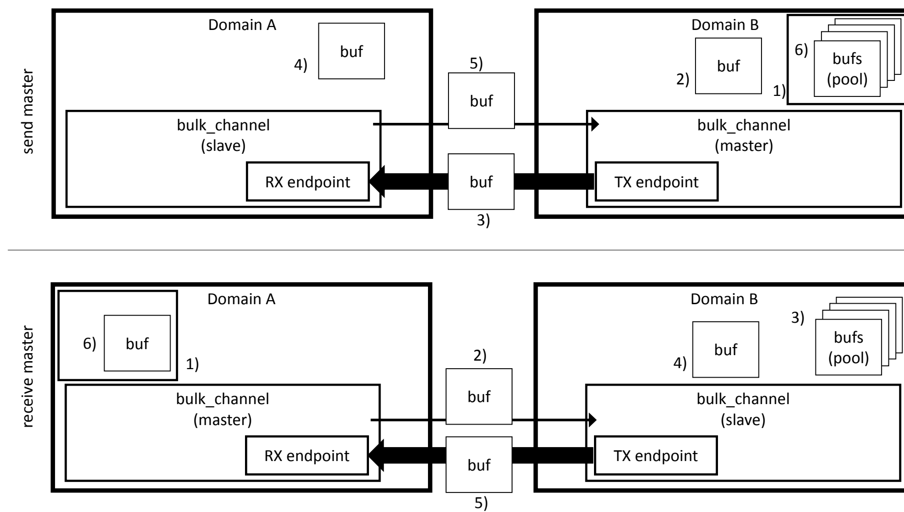

Figure 5.4: Bulk Transfer Architecture

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 39 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 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.

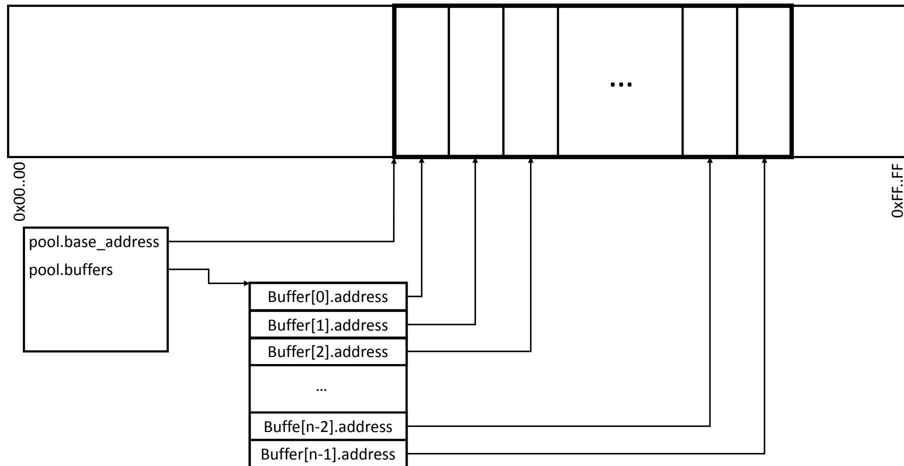

Figure 6.1: Pool and Buffer Representation

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

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.

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

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

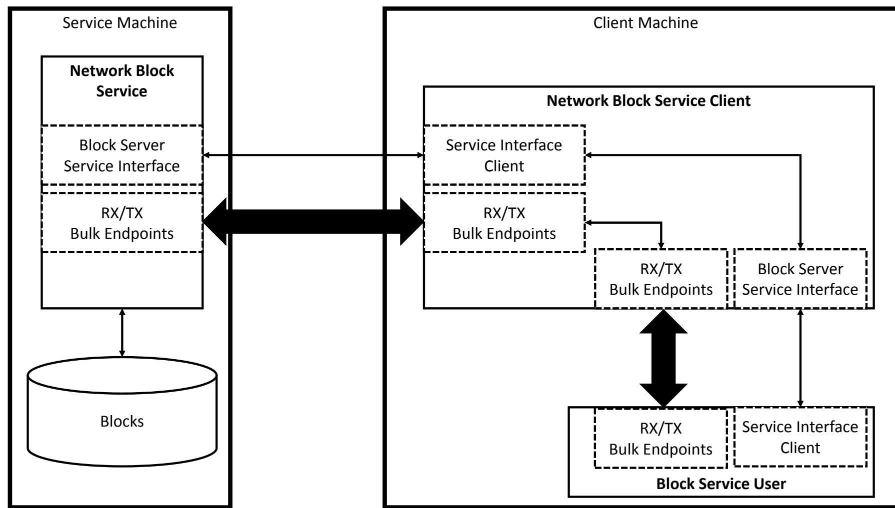

Figure 7.1: Network Proxy using Local Channel

- **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 [8] 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 [**?**]. This integration can lead to a unified way of exchanging messages of different size between domains and even machines.

Part II

Shared Memory Bulk Transfer

Table of Contents

1 Introduction **57**

2 Related Work **58**

2.1 fbufs 58

2.2 rbufs 58

3 Design Considerations **60**

3.1 General Design Challenges 60

3.2 Capability System 60

3.3 Using Capabilities For Bulk Buffers 61

3.4 Trusted Third Party 61

3.5 New Capability 62

3.6 Limitations 64

4 Implementation **65**

4.1 Existing Infrastructure 65

4.1.1 Libbarrelfish Bulk Transfer 65

4.1.2 Flounder 65

4.2 Backend Implementation 66

4.2.1 Overall Design 66

4.2.2 Channel Creation 66

4.2.3 Transferring Buffers 67

4.2.4 Implementation Challenge: Message Ordering 68

4.2.5 Implementation Challenge 2: Unnecessary Locking Of Monitor 70

5 Evaluation **72**
5.1 Bulk Transfer Semantics 72
5.2 Bulk Transfer Performance 74
5.2.1 Setup 74
5.2.2 Testing Hardware 75
5.2.3 Results 77
5.2.4 Implications For Users 82
5.2.5 Possible Improvements 82

6 Future Work **83**

List of Figures

3.1 State diagram of a single shared_frame capability. 63
4.1 Messages transmitted between channel endpoints in a copy operation. 68
5.1 Interaction of the two agents in the `bulk_shm` test. 73
5.2 Microbenchmark configuration with communicating domains on the same core. 76
5.3 Microbenchmark configuration with communicating domains on different cores. 76
5.4 Microbenchmark configuration with additional hops between communicating domains, all on different cores. 76
5.5 Latency as a function of bulk buffer size. 78
5.6 Latency for different core configurations. 79
5.7 Throughput for different core configurations. 79
5.8 Latency for different core configurations. 81
5.9 Throughput for different core configurations. 81

Chapter 1

Introduction

Using shared memory to send large buffers is not a new idea. In many systems, shared memory has been the main form of inter-process communication. Using the CPU to copy buffer contents from one memory location to another is expensive, and has only become more so as the gap between processor frequency and memory latency increased.

In traditional SMP systems, much has been done to defer the actual copying operation until it is absolutely necessary (e.g. copy on write), and services like the network stack use copy-free message systems. These message systems traditionally do not offer any isolation between the sending and receiving side, so they require a certain amount of trust. They also have very specific interfaces that have very little in common and directly reflect the specifics of their respective implementations. In systems like Barrelfish, where consistent shared memory is optional and core services run as regular userspace processes, it is much more desirable to make all communication explicit through message passing.

We designed and implemented a system for sending large messages – an abstraction for sharing memory with no unnecessary copying. The common abstraction of this bulk transfer interface, that also can be used to send bulk messages across an actual network using the exact same calls, was detailed in chapter 6.

In this part, we discuss how the shared memory backend was designed and implemented, as well as some performance measurements and their implications. The target of our implementation is a modern multiprocessor with cache-coherent shared memory. While it will also run on NUMA machines, we did not design it with that environment in mind.

Chapter 2

Related Work

Several influential designs for bulk transfer mechanisms were already presented in chapter 2 as they guided the design of the common interface from the beginning. That these systems all used shared memory for their implementation is simply due to the fact that traditional SMP systems (as well as many unconventional ones) are heavily grounded in shared memory communication. In this chapter, we explore the designs again, but with an emphasis on the optimizations they employed to boost performance.

2.1 fbufs

The article discusses several methods of increasing the speed when sharing immutable buffers. It argues that in many cases, data travels the same path along domains. This is especially true when looking at network traffic, where inspecting all the packet headers clearly determines the intermediate and target domain of the packet payload. This is referred to as an *I/O data path*. The article suggests to keep page mappings of buffers valid even if the buffers are no longer needed, and to reuse them for more traffic on the same path. This is to avoid traps into the kernel and locking of VM data structures. It goes even further and suggests not enforcing access policies on buffers for higher throughput. It points out at the end that this is at the cost of security.

The influence of these thoughts are mainly twofold. First, we note that prior knowledge of I/O data paths is not a given when looking at general bulk transfer, as opposed to the specific problem of network traffic studied in the article. This idea however was integrated with the concept of the *bulk pool* in our bulk transfer facility. The same pool should be used over multiple bulk channels for the same bulk transfer data path, allowing to reuse existing data structures and possible mappings. Second, the choice to go for increased throughput at the cost of isolation was emphasized from the beginning of the design stage and resulted in the different trust levels for the channels and pools involved in the bulk transfer.

2.2 rbufs

The Nemesis system was intended for the specific purpose of providing quality of service guarantees to multimedia applications. This can be seen in the deci-

sion to have applications do their own processing of work traditionally done by central services in order to have scheduling control, as well as in the optimization of having a single virtual address space for all processes to share, so that context switches would have a smaller impact on the TLB and cache.

Nemesis actually has many similarities with Barrelfish, as both try to push as much functionality as possible into the applications themselves via libraries, with privileged userspace services providing the parts that regular applications can not be trusted with. In both systems, the kernel's main purpose is scheduling, as it has the unique ability to preempt all other code.

Rbufs partially inspired our early desire to support aggregate objects on the lowest level, before we later decided to rather have our interface be generally extensible instead. While the rbufs design is a clean approach to fully trusted channels, it is unclear how it would be extended to support isolation without getting rid of the main performance optimization.

Chapter 3

Design Considerations

3.1 General Design Challenges

The main challenges for our shared memory backend design were the following:

- Buffer contents should not be copied if it can be avoided.
- If both domains trust each other, they should not pay a performance penalty for using our system.
- If the domains do not trust each other, they should have enforcable isolation guarantees.
- The system should fit into Barrelfish's distributed design.

The last two points are especially important, because they explain why other bulk transfer approaches, such as fbufs or rbufs (see chapter 2) can not just be adapted to our system.

In the message passing abstraction of our interface, doing a *move* operation conceptually makes the buffer go away, and the other side will expect us to now leave it alone. The traditional shared memory bulk transfer systems were not made with isolation in mind, as their primary purpose was communication between mutually trusted subsystems.

3.2 Capability System

In Barrelfish, all physical memory is managed in userspace by the applications themselves. To insert a mapping into its page table, an application needs to have a capability for the physical memory. To share memory with other domains, applications can send each other their capabilities. While the kernel and monitor enforce the rules of the capability system, the decision of when to use which capability is purely the application's. A comprehensive analysis of the capability system in Barrelfish both as it is and as it is intended to become can be found in Mark Nevill's master's thesis[7].

Because all capabilities are stored in a protected area (called cspace), capabilities can also be revoked: all copies of a capability – no matter what domain they belong to – are deleted, and only the copy that has been revoked remains.

This is a very powerful operation, as it gives guarantees about the non-existence of other capabilities.

Any protocol based on sharing memory in Barrelfish must necessarily involve the domains exchanging capabilities. The big advantage of this approach is that the kernel does not need to understand the semantics of the protocol – it merely enforces the semantics of the general capability system.

3.3 Using Capabilities For Bulk Buffers

If there are no isolation requirements (i.e. the domains trust each other), then all that is needed for bulk transfers is for both domains to map the same capability into their virtual address space. Since they never have to unmap anything, it is sufficient for them to share just one big capability for the whole buffer pool, which can be sent along with a pool assignment message.

If the domains do not trust each other, we do not want them to be able to overwrite data they have sent. In the case of a *move* or *pass* operation, only the receiver will need access to the data, so we can simply remove the capability from the sender's cspace. Because the receiving domain does not trust the sender to delete the capability voluntarily, it has to revoke the capability it received.

The biggest challenge is the *copy* operation, because it involves more than one domain having access to the data. Both the sender and the receiver want a guarantee that the other side can not overwrite the data, which means that neither of them should have a read-write capability. However, as soon as both have released the copy, we want the owner to regain full read-write access.

3.4 Trusted Third Party

One way to give the isolation guarantees for untrusted copy operations would be to introduce a trusted third party, a userspace process with the specific purpose of managing *copy* operations. To send a *copy* message, the owner of the buffer would transfer ownership to this third party, which would then revoke the capability and give both sides a read-only capability. When the buffer is released, the third party transfers ownership back to the sender. This approach addresses both issues of copying: Both domains know that the other can not overwrite the data, and the owner can regain its privileges after the buffer is released. However, introducing such a third party also has serious drawbacks:

- Both domains need to trust another service.
- The trusted service needs to manage all channels it is involved in.
- *Copy* messages are essentially relayed, increasing latency.
- If there is only one such trusted third party, it could quickly become a bottleneck.

The connection to the service could either be separate from the bulk channel, in which case the service would also need to authenticate requests for buffer access, or it could be inserted between the sender and receiver as a relay station, increasing latency for all messages (not just copies). For these reasons of

scalability, we would prefer to extend the existing trusted infrastructure instead – the capability system.

3.5 New Capability

As the existent capabilities do not provide the functionality we need to send untrusted *copy* messages, we would like to propose a new one. Our new capability must have the following semantics:

- Represent a mappable region of physical memory
- There can be multiple read-only copies
- There can only be one read-write copy
- There can not be read-only and read-write copies at the same time
- There must be a way to regain a read-write copy if there are no read-only copies left

Ideally, the capability should also largely behave like the existing frame or device-frame capabilities to better integrate into existing code.

These requirements are analogous to the classic multiple readers / single writer lock, but need to be enforced by a distributed system. While the current Barrelfish capability system does not contain any capabilities with comparable semantics, it can be extended using a simple state machine. Figure 3.1 shows a simple state diagram that is sufficient to satisfy all our requirements.

The most important insight is that while the semantics are stating invariants on the global state, the transitions between reading and writing can only happen when there is just one such capability in the whole system. This allows us to encode a state machine directly into the capability and keeping all the copies' states consistent without ever having to change the state of more than one copy.

Just by enforcing that such a capability can not be copied or sent while in the exclusive state, and that the only way to get back to the exclusive state is revocation, we can guarantee that there will never be both an exclusive and a shared copy simultaneously in the whole system.

This requires some additional checks in the invocation handlers in the kernel, but these are very simple (reading the capability type and one of its fields) and – most importantly – completely local. Invocations like copy and send can already fail and return error codes to the user (e.g. if the invocation targets a non-existent cspace slot), so it is not a big step to add more failure conditions. The invocation that changes the state from exclusive to shared should check if the capability has already been mapped, and if so either remap it read-only or return an error to the user.

Simply possessing a read-only copy of such a capability guarantees a domain that there is no corresponding read-write capability in the whole system. Another noteworthy aspect is that this capability allows us to regain rights that were previously reduced, without any outside intervention.

A more subtle aspect of this new capability is that it can not be generated by retyping a frame or device frame. This decision was made, so that there would not be any mappable ancestors – if one revokes a frame that was retyped

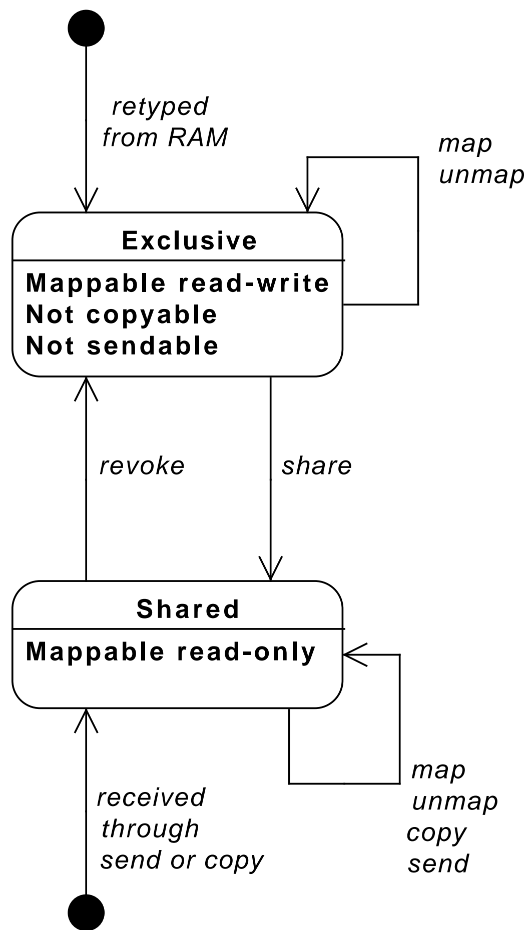

Figure 3.1: State diagram of a single shared_frame capability.

from a bigger frame, the bigger frame does not get deleted, so the holder could still access the same memory. Retyping directly from RAM eliminates this possibility, as RAM capabilities can not be mapped, and a capability can not be retyped twice.

It is important to note that this shared_frame capability not only provides the semantics we want for the untrusted *copy* operation, but it also does not interfere with untrusted *moves*. This means we can use it instead of regular frames for untrusted pools without having to differentiate between movable and copyable buffers.

For trusted channels, we still want to use regular frames, because we need to be able to have the same buffer mapped read-write in both domains.

3.6 Limitations

Due to time constraints, we could not actually implement the shared_frame capability (as it only provides security guarantees, more essential functionalities were prioritized). The current Barrelfish capability system differs from the one specified by Nevill[7] mostly in that the remote capability database is deactivated, so the monitors do not manage the global capability space. Revocations across cores are currently just ignored, which means that we can not actually give the security guarantees in the untrusted case until the system is reactivated.

Chapter 4

Implementation

4.1 Existing Infrastructure

4.1.1 Libbarrelfish Bulk Transfer

There already is a simple bulk transfer mechanism implemented in libbarrelfish, the Barrelfish core library[11]. However, that implementation has several important limitations:

- Both domains have full read-write access to all buffers at all time, so they have to trust each other not to overwrite data the other side is reading.
- The interface is explicitly machine-local, so it does not provide a message passing abstraction.
- There is no standardized way to share buffers between more than two domains. It can still be done on top of the interface, but the user has to manage it.

A similar system called *pbufs* also exists, and it shares these drawbacks. For a more detailed analysis see section 3.4.

In order to be able to give strong isolation guarantees and provide a common interface with the networking backend, we did not use the existing bulk transfer code.

4.1.2 Flounder

As Barrelfish is a purely message based system, there is a common interface for the different ways that messages are implemented in the backend. A domain specific language called Flounder[3] is used to specify interfaces, defining the structure of messages that can be sent and received. Depending on where the communicating domains are actually located at runtime, the appropriate message passing backend is then chosen to send such messages.

LMP

If both domains are running on the same core, then local message passing (LMP) is used. Data is transmitted by loading it into registers, invoking a capability

for the other domain and having the kernel upcall the receiver with the data from the registers.

UMP

If the domains are running on different cores, but have access to shared memory, then user-level message passing (UMP) is used. The sender copies the message data into a shared region of memory, and the receiver then copies it into a local buffer. To transmit capabilities, the sender has to first send it to its local monitor, which will transmit it to the other core's monitor, where it will be inserted into the receiver's cspace.

Flounder allows messages to contain fixed-size or variable-size arrays, so it could be used to directly send bulk messages. However, both backends are designed to only transmit small chunks at a time (about 10 registers worth of data) and copy the message payload.

4.2 Backend Implementation

4.2.1 Overall Design

We implemented a backend to the bulk transfer interface specified in chapter 6 to be used for bulk transfer on a cache-coherent machine in a symmetric multiprocessing setup. As shared memory can always be used in such a setup to make data available to another domain, our backend implementation could avoid physically copying or moving bulk data in memory. The backend's task therefore is twofold: First, it needs to ensure the proper memory mappings of the buffer in each involved domain according to the access rights and trust levels specified in section 5.2. Second, it needs to notify the other endpoint about buffers being transferred and allocate and manage the corresponding data structures for the library representation.

For communication amongst the domains, we used Barrelfish's message passing infrastructure provided by Flounder. A Flounder channel is established between to the bulk transfer endpoints. The channel is used to notify the peer endpoint about operations on the channel as well as transmitting corresponding metadata and buffer capabilities.

4.2.2 Channel Creation

Our shared memory bulk transfer backend is tightly coupled with the Flounder communication channel. For a Flounder channel, one side *exports* the channel interface and receives a unique identifier, the `iref`. The interface our Flounder channel uses is specified in `if/bulk_ctrl.if`. The other side can then *bind* to that `iref` to establish the channel. From then on, messages can be exchanged.

Channel creation for the bulk transfer library runs in two phases. First, the underlying Flounder channel is established. Second, messages are exchanged to verify whether the bulk channel properties of the two endpoints match: channel role and channel trust. This phase is called *negotiate*.

The bulk channel endpoint descriptor for shared memory contains only the `iref` of the underlying Flounder channel. Two functions for endpoint creation

are available[1]: `bulk_sm_ep_create` and `bulk_sm_ep_create_remote`. The first creates an endpoint to be used with `bulk_channel_create`. It exports the interface. The second creates an endpoint to be used with `bulk_channel_bind`. It takes as argument the interface's `iref` which the bind function will use to establish the Flounder channel. To establish a bulk channel between to domains, the bulk channel's `iref` can either be registered to the name service or conveyed by some higher-level negotiation protocol that uses bulk transfer as a sub-mechanism.

There is one caveat to this procedure: Exporting an interface binds it to a waitset to dispatch events. By the design of our bulk library, the waitset is specified at channel creation, not endpoint creation. As a result, the endpoint's `iref` is only valid after the call to `bulk_channel_create` and must not be communicated to the other domain before that. The library will verify this is true and report an error otherwise.

4.2.3 Transferring Buffers

If the user wants to *copy* or *move* a buffer, he calls the library and registers a continuation to be notified about the result. The sequence of events generated by the shared memory backend are depicted in Figure 4.1 for a *copy* operation. A Flounder message notifies to other endpoint about the buffer. The other side generates the corresponding library data structures and performs an upcall to the user-defined `buffer_copied` callback.

No buffer data is actually transferred – the contents of the bulk buffer are never accessed by the library. If the channel is trusted, the whole memory region of the buffer's pool are already mapped in the domain's address space. In the untrusted case, the corresponding frame capability is transferred with the message and mapped. Two implementation issues found communicating with Flounder are described in the next two sections.

[1] see `include/bulk_transfer/bulk_sm.h`

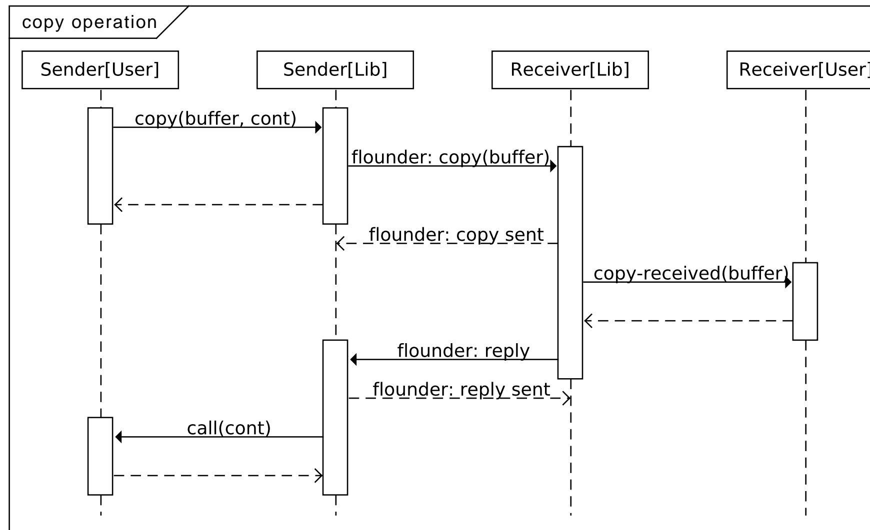

Figure 4.1: Messages transmitted between channel endpoints in a copy operation.

4.2.4 Implementation Challenge: Message Ordering

Using Flounder channels, sending messages may fail with a temporary `FLOUNDER_ERR_TX_BUSY` error. In such a case, the message queue is full and the user needs to retry sending the message at a later point in time. Such errors occur often when using the bulk transfer library. There are several implications of this behaviour for the bulk transfer implementation. The issues are presented looking at a *copy* operation as seen in Figure 4.1.

For the implementation, receiving a temporary transmission error is fine if the send request comes directly from the user, e.g. the buffer copy request cannot be transferred. The error can be reported to the user. However, this is not always true. On the receiver side, the user is not actively making calls, but gets notified about the new buffer by a registered callback. He is not involved in sending the reply back to the receiver. The error cannot be reported. As this is a common case, Flounder allows to register a closure to be called as soon the message can potentially be transmitted. Thus, retransmission can be triggered without user involvement. The general approach in the Flounder example codes is to split sending a message into two functions. The first is to prepare all the arguments required for the message and store them on the heap. The second function attempts to send the message using the arguments prepared and frees the data upon success. Using this approach, the second function can directly be registered as resend closure. This said, one problem remains: Only one resend closure may be registered at any time. Registering further resend closures fails. As multiple bulk transfer operations can be pending on the same bulk channel, this is not enough.

To our knowledge, Barrelfish does not provide means to deal with this sit-

uation. Our first solution was to introduce a thread-safe resend queue to the bulk channel. The sending function (the second of the two functions mentioned before) was structured as follows: It would try to send the message. If sending failed, it would insert itself into a resend queue. The resend function was designed as a general purpose function that would dequeue the first element of the resend queue and call the closure. If sending failed again, the sending function would reinsert itself into the resend queue as in the attempt before. This approach guaranteed that all messages would be sent eventually. However, it resulted in arbitrary message reordering leading to an unexpected and impossible communication pattern from the user perspective.

First, this led to a race condition on the flounder channel send queue. A message attempted to be sent as a direct consequence of a user call to the library was reported back to the user. He would retry the library call. At the same time, reply messages by the library that could not be sent were inserted into the resend queue. When the resend handler was called, retransmitting the message could fail because the send queue was already filled again by a user library call. As a result, a message generated by a user library call could have been transmitted before a reply message due to a previous event, leading to message reordering.

Secondly, the bulk channel resend handler would dequeue a closure from the channel's resend queue and call it. The closure would reinsert itself into the queue if sending the message failed. However, it would reinsert itself at the end of the queue. Thus, closures inserted into the queue at a later time could be sent before the response to the message that triggered them in the first place, leading to the user being notified about events in an acausal (and largely random) order.

In this setup, we could observe incorrect interaction such as this: Agent A would copy a buffer to an agent B. As depicted in Figure 4.1, the library will automatically send a copy reply back to agent A. However, transmission of the copy reply fails and it gets added to the resend queue. Meanwhile, agent B would happily process the data and decide to release the buffer again. Sending the message succeeds. As a result, agent A will receive a *buffer released* message before knowing the buffer was actually copied successfully. Even worse, agent A will receive the *buffer copied successfully* message long after the buffer was already released, about which he was already notified. Clearly, the bulk transfer implementation could not be used meaningfully in such a configuration.

We made two changes to this first approach, to achieve FIFO ordering of messages on the Flounder channel. First, we changed the usage policy of the two send functions present for every Flounder message we defined. Recall that the first function is to prepare the message arguments on the heap and call the second function. The second function is to attempt to transmit the message and insert itself into the bulk channel resend queue upon failure. In the new implementation, the second function no longer serves as a Barrelfish send closure. It only attempts to send the message and returns an error code about the success. The first function no longer calls the second but directly inserts it into the resend queue. The resend handler again looks at the first element in the resend queue and calls the function. However, it only dequeues the element if sending the message actually succeeds. This prevents reordering of messages in the resend queue. Secondly, even messages attempted to be sent as a direct result of a user library call are now inserted into the resend queue. Thus, the race condition on the flounder channel send queue is removed. This yields FIFO

ordering of all messages on the Flounder control channel. The corresponding functions are implemented in `flounder_helpers.c` of the shared memory bulk transfer backend.

One difficulty our automatic resend handling still has is that it requires the dispatching of waitsets by the user, who may not be aware that his message will only really leave the domain if he continues dispatching. While it is reasonable to assume that the user would dispatch the waitset anyway – in anticipation of a reply – there might be edge cases where he would not expect to still need to dispatch events (e.g. if he just got what he knows to be the last *pass* operation, but never dispatches the following event that would send a reply back to the other side).

4.2.5 Implementation Challenge 2: Unnecessary Locking Of Monitor

Using a Flounder channel for our backend messages allowed us to abstract away the topology of CPU's. Depending on where the two communicating domains are located, Flounder will automatically use the appropriate form of data transfer. However, while the different Flounder implementations all provide the same functionality, they differ drastically in their performance when transferring capabilities.

LMP If the receiver is on the same core, then sending a capability is a local operation. The message is transferred by the kernel, which manages the cspace for both domains. Sending a capability is no different than copying it locally, and does not require any synchronization. Furthermore, messages with capabilities are treated exactly the same as messages without, so there is no penalty for defining messages with optional capabilities.

UMP If the receiver is on a different core, then sending a capability is not local anymore and can not be done by the kernel alone. Instead, the request is first sent to the local monitor, which communicates with the monitor on the other core to place a copy of the capability into the receiving domain's cspace. As the capability takes a different path than the rest of the UMP message payload, the receiver has to wait for two different receiving events to occur before it can reassemble the complete message. Messages without capabilities on the other hand are directly transmitted over shared memory, involving neither the kernel nor monitor.

If one really needs to transmit a capability, then the UMP performance penalty is easily justifiable as the price of a consistent capability system. However, if one sends a message that could contain a capability, but doesn't, then the UMP backend will still connect to the monitor, acquire a mutex and send a `NULL_CAP` – leading to the same performance penalty.

In our bulk transfer backend, the main distinction between trusted and untrusted channels is that we only send the individual buffers' capabilities around in the untrusted case. If the channel is trusted, we once send a capability for the whole pool, and only refer to offsets within it on all subsequent move, pass, and copy operations. We first wanted to use the same Flounder messages for both trusted and untrusted operations, with the capability simply being set to `NULL_CAP` by trusted channels. As a result, trusted channels over UMP were

almost as slow as untrusted ones, since the monitor synchronization overhead dominated all messages (the untrusted case was still slightly slower, because it also performs two additional invocations on each receive). To get rid of this unnecessary synchronization, we had to define separate messages for the trusted case. This simple change sped up trusted UMP channels by a factor of 20. Bulk channels over LMP were unaffected.

Chapter 5

Evaluation

5.1 Bulk Transfer Semantics

To verify the correct functioning of the shared memory bulk transfer implementation as well as the buffer access semantics, the `bulk_shm` application was developed. It can be found in `usr/tests/bulk_transfer/bulk_shm.c` in the Barrelfish source tree. It implements the role of two agents that run in separate domains and establish a bulk transfer channel. Their communication is outlined in Figure 5.1. A total of seven different tests can be run.

Tests 0 and 1 are used to verify that sent buffers really do contain the data they are expected to on the receiver's side. They run the communication depicted in Figure 5.1 omitting other tests. A bulk channel is established and buffers are copied as well as moved on it. In particular, the following functions are covered in the test with the shared memory backend in use.

- `bulk_sm_ep_create`
- `bulk_sm_ep_create_remote`
- `bulk_channel_create`
- `bulk_channel_bind`
- `bulk_channel_assign_pool`
- `bulk_channel_pass`
- `bulk_channel_move`
- `bulk_channel_copy`
- `bulk_channel_release`

In test 0, a trusted bulk transfer channel is used. In a trusted scenario, all the buffer frames are mapped in each of the endpoints' domains as soon as a pool is assigned. The underlying Flounder channel is used solely to notify the receiver about buffers being transferred.

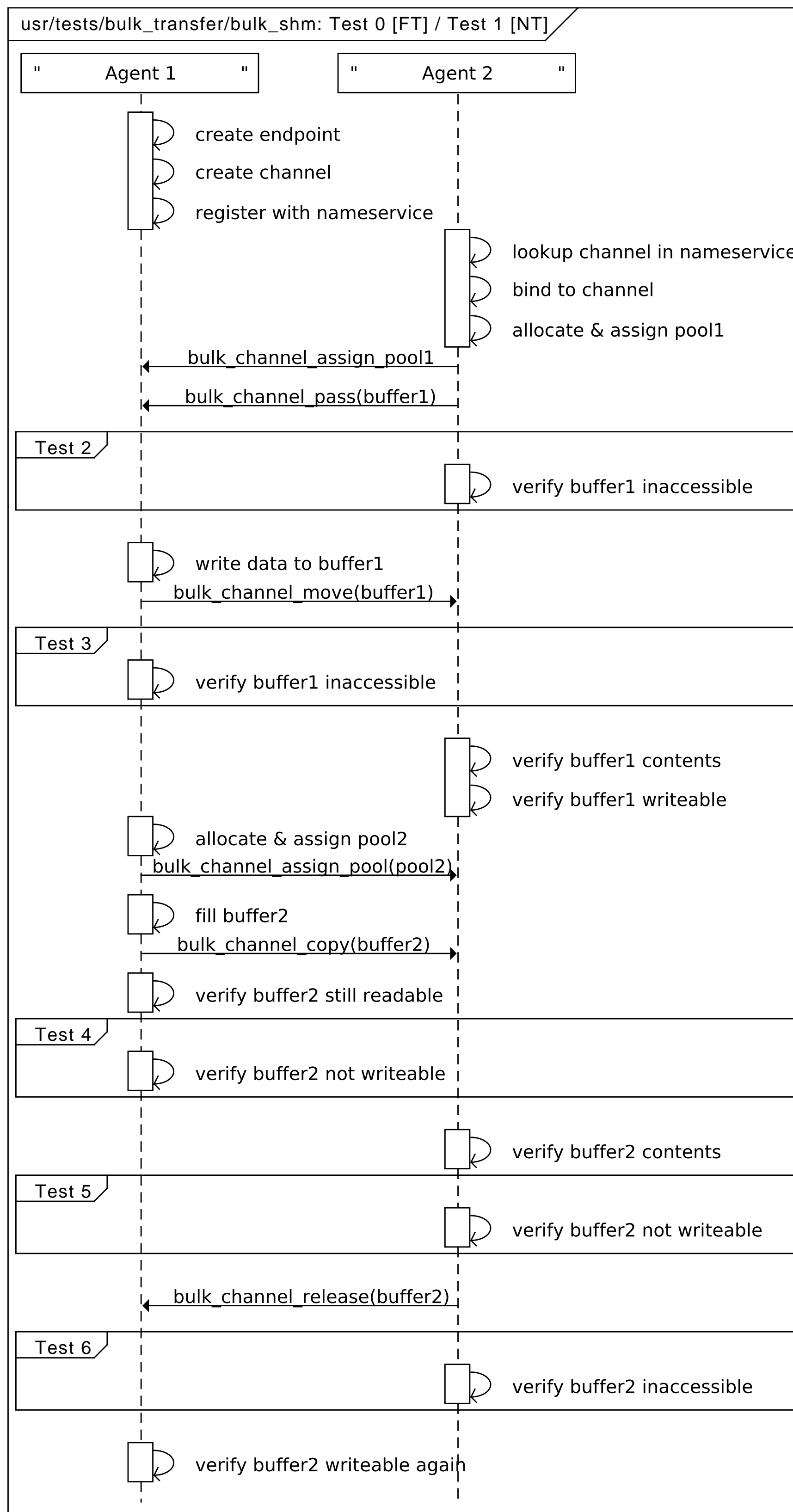

Figure 5.1: Interaction of the two agents in the `bulk_shm` test.

Test 1 establishes a non-trusted bulk transfer channel. In the non-trusted scenario, frame capabilities for the buffers being copied or moved need to be transferred along with the notifications about buffer transfer. Tests 0 and 1 ensure the implementation provides the functionality specified and that the data being transferred is received correctly.

The access rights to buffers has been specified in section 5.2. If domains cannot expect their communication partner to adhere to these conventions, they need to be enforced. To what extend they can actually be enforced in the capability system currently provided by Barrelfish and possible design alternatives have been discussed in section 3.1. Tests 2 through 6 of `bulk_shm` verify the guarantees our implementation provides. These include

Test 2 A *passed* buffer can no longer be accessed.

Test 3 A *moved* buffer can no longer be accessed.

Test 4 A buffer shared by a *copy* operation is read-only.

Test 5 A buffer received by a *copy* operation is read-only.

Test 6 A *released* buffer copy can no longer be accessed.

The agents establish a non-trusted bulk transfer channel and start interacting as in the first two tests. At the step indicated in Figure 5.1, one of the agents will try to break the guarantee specified in the test. The test is passed if the agent's domain produces a page fault. The other domain will point out it expects it's peer to fail and enter an endless loop.

The `bulk_shm` application takes two commandline arguments. The first defines which agent's role to take on (1 or 2) and the second defines the test to run (0 – 6). This can be specified as follows in the Barrelfish boot configuration `menu.lst`:

```
# bulk_shm Test 0 in dual-core setup
module /x86_64/sbin/bulk_shm core=0 1 0
module /x86_64/sbin/bulk_shm core=1 2 0
```

Listing 5.1: `bulk_shm` boot configuration

Our shared memory bulk transfer implementation passes all seven tests. This is true regardless of whether the agents run on the same or on different cores. As we know, there is no revocation across cores in the current capability system, so this test is clearly quite limited in scope: neither agent goes out of its way to violate the guarantees. It does however show the view of a regular user of the library.

5.2 Bulk Transfer Performance

5.2.1 Setup

To evaluate the shared memory bulk transfer implementation, the benchmarks and block service developed in collaboration with the network bulk transfer team were used. These microbenchmarks consist of an echo server (that just sends back every buffer it receives) connected to either a throughput or latency

load. The latency benchmark sends a single buffer, and measures the time until the echo is received. The throughput benchmark sends many buffers in quick succession and measures the time it takes until the last echo is received. These benchmarks can be configured to use different buffer sizes and sample sizes as well as the following more backend specific options:

Trust As trusted channels never unmap their buffers, and do not need to send capabilities along with every buffer operation, we would expect them to perform much better than untrusted channels.

Core Residence Depending on whether the communicating domains are running on the same core or not, the Flounder channel that the shared memory implementation is built on uses either the LMP or UMP backend.

Intermediate Domains In a real-world scenario, we expect bulk transfers to span multiple domains such as for example an application, service provider and driver. To simulate this situation, the `bulk_transfer_passthrough` application was developed that can be inserted between any two communication endpoints to study the influence of additional hops in the communication path.

The different communication scenarios we set up are illustrated in Figure 5.2, Figure 5.3 and Figure 5.4 respectively.

The boot configuration we used is provided in Listing 5.2 for the latency microbenchmark. The `micro_rtt` binary can be replaced by `micro_throughput` for throughput testing.

```
# RTT microbench single-core
module /x86_64/sbin/bulkbench_micro_echo sml:bench1 sml:bench2
module /x86_64/sbin/bulkbench_micro_rtt smc:bench2 smc:bench1

# RTT microbench dual-core
module /x86_64/sbin/bulkbench_micro_echo core=0 sml:bench1 sml:bench2
module /x86_64/sbin/bulkbench_micro_rtt core=1 smc:bench2 smc:bench1

# RTT microbench multi-core (trusted)
module /x86_64/sbin/bulkbench_micro_echo core=0 sml:bench1_2 sml:bench2_1
module /x86_64/sbin/bulk_transfer_passthrough core=1 bench1_2 bench1_1 1
module /x86_64/sbin/bulk_transfer_passthrough core=2 bench2_1 bench2_2 1
module /x86_64/sbin/bulkbench_micro_rtt core=3 smc:bench2_2 smc:bench1_1
```

Listing 5.2: Boot configurations for microbenchmarks.

5.2.2 Testing Hardware

All measurements were taken on the ziger2 machine, which has a cache coherent AMD Istanbul architecture with a total of 24 cores in 4 sockets. All cores are clocked at 2400 MHz. This is a fairly standard SMP design, so we expect our results to translate to other SMP machines as well.

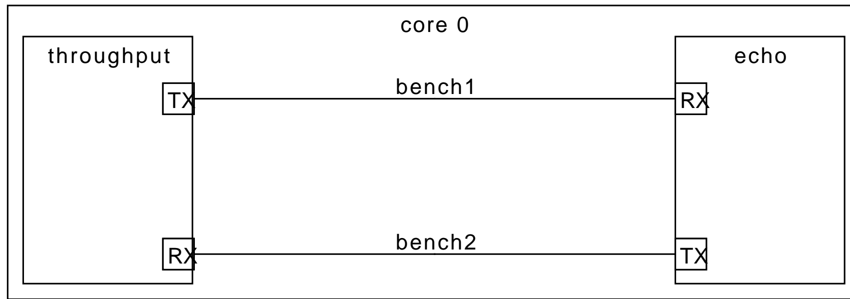

Figure 5.2: Microbenchmark configuration with communicating domains on the same core.

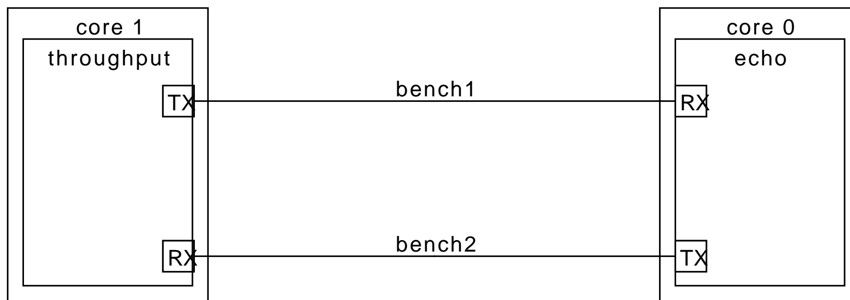

Figure 5.3: Microbenchmark configuration with communicating domains on different cores.

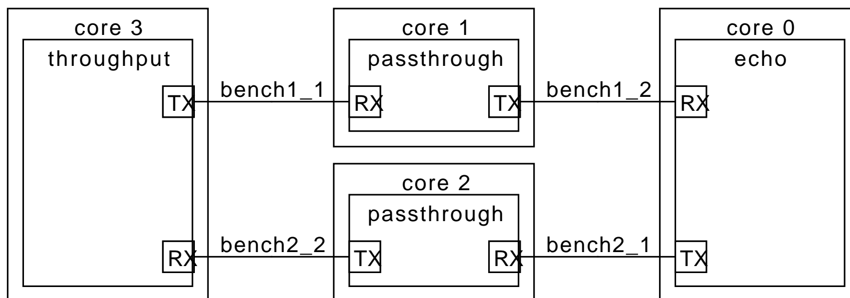

Figure 5.4: Microbenchmark configuration with additional hops between communicating domains, all on different cores.

5.2.3 Results

Buffer Size

As it can be seen in Figure 5.5, the size of the bulk buffers has virtually no influence on the latency of operations. This is because our implementation never even accesses the buffers themselves, but only sends messages with references and capabilities – and neither the number of messages nor their payload size changes with buffer size. The very small increase in latency seen in the graph is probably caused by the fact that in the untrusted case, these much larger regions of memory need more page table entries to be changed with every mapping and unmapping. We did not take detailed measurements for the trusted case, because it is even less dependant on buffer size: the memory is only mapped once on pool assignment, and never unmapped.

This constant latency for variable buffer size makes it almost meaningless to talk about general maximal throughput, as it is only bounded by what the user can process. On the other hand, it is unrealistic to expect users to choose megabyte-sized buffers. To measure throughput in a meaningful and comparable way, we decided to use the rate of transmitted buffers, independent of their size, for all following benchmarks.

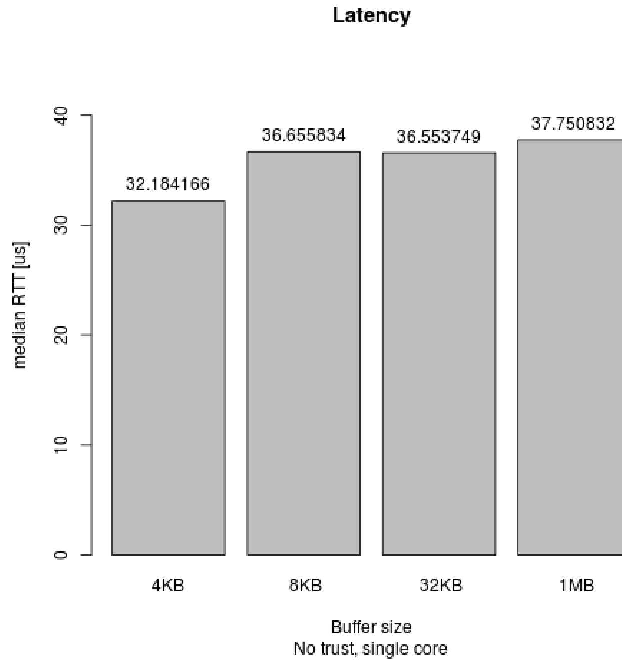

Figure 5.5: Latency as a function of bulk buffer size.

Core Configuration – Trusted Channels

When comparing different configurations of trusted channels, there is a remarkable difference between throughput and latency, as seen when comparing Figure 5.6 and Figure 5.7. While a single-core setup has the smallest latency, it's throughput is worse than if two domains were running on separate cores. This is probably because an individual message benefits from LMP transfers automatically scheduling the receiving domain and notifying the dispatcher of the new message, whereas UMP channels need to be actively polled by a thread. If the channels are however flooded with messages (as in the throughput benchmark), then the dual-core setup benefits from being able to execute both domains at the same time and sending messages without leaving userspace, whereas the instant delivery mechanism of LMP becomes a burden, since it requires a context switch for every single message.

Surprisingly, doubling the number of UMP hops does not double the latency. We do not know why it reduces throughput by a factor of 20, as this is a very extreme result that even memory allocation problems could not account for. As UMP relies heavily on cache-coherence for its efficiency, we can only speculate that the flood of memory accesses to pairwise shared cache lines could be responsible.

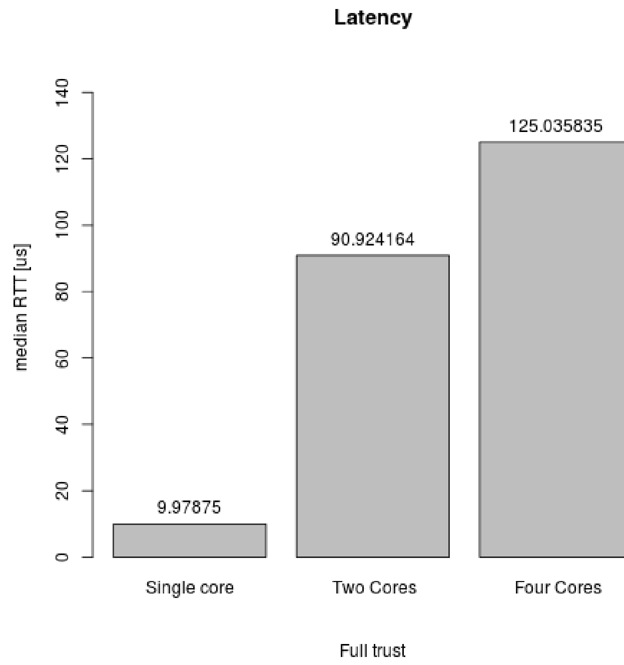

Figure 5.6: Latency for different core configurations.

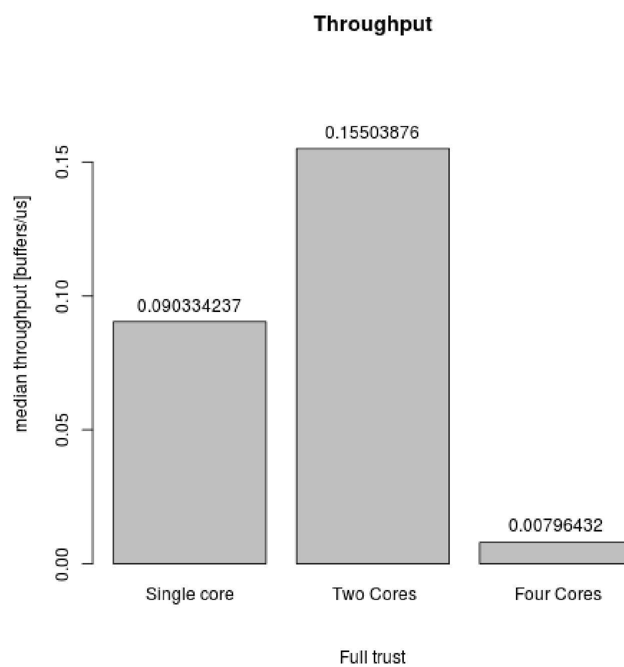

Figure 5.7: Throughput for different core configurations.

Core Configuration – Untrusted Channels

As was to be expected, untrusted channels are much slower than their trusted counterparts, since they are sending capabilities around with every move, and then revoke and map them once received. In contrast to the trusted case, Figure 5.8 and Figure 5.9 both show the same ranking for the different configurations. Single-core setups are unsurprisingly by far the fastest, since they can be served locally and LMP messages go through the kernel anyway.

The much slower transmission across cores can be explained by how UMP works: the sender first has to acquire a lock on the local monitor (using LMP), then send the capability to that monitor (using LMP). The monitor serializes the content of the capability (using a system call), and sends this data to the monitor on the other core (over UMP). That monitor then generates a new capability from the serialized data and sends it to the receiving domain (using LMP). So every single capability transmission over UMP uses several LMP messages and additional system calls.

As the remote capability database was still deactivated at the time of our measurements, they actually understate the true cost. In the fully operational system, revoking capabilities across cores will be significantly more expensive, as it will involve connecting to the monitors and having them communicate with each other as well. This enormous performance penalty across cores is the price of a consistent distributed capability system – any other bulk transfer implementation that sends capabilities over UMP would be just as slow.

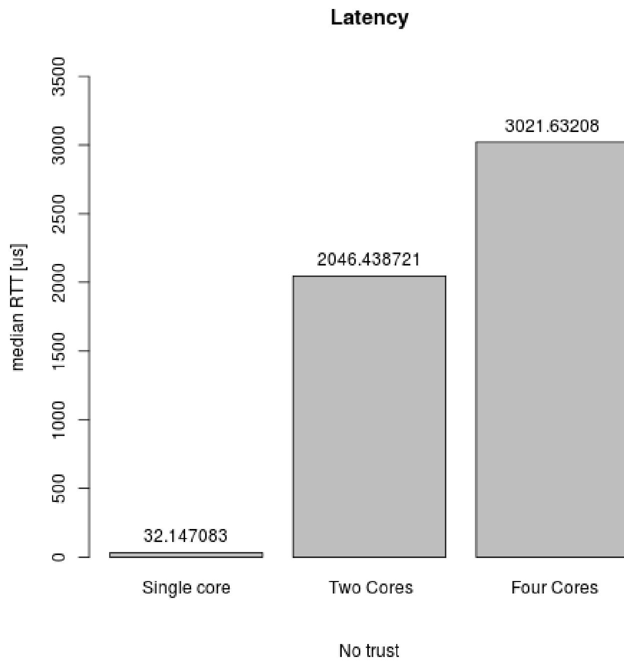

Figure 5.8: Latency for different core configurations.

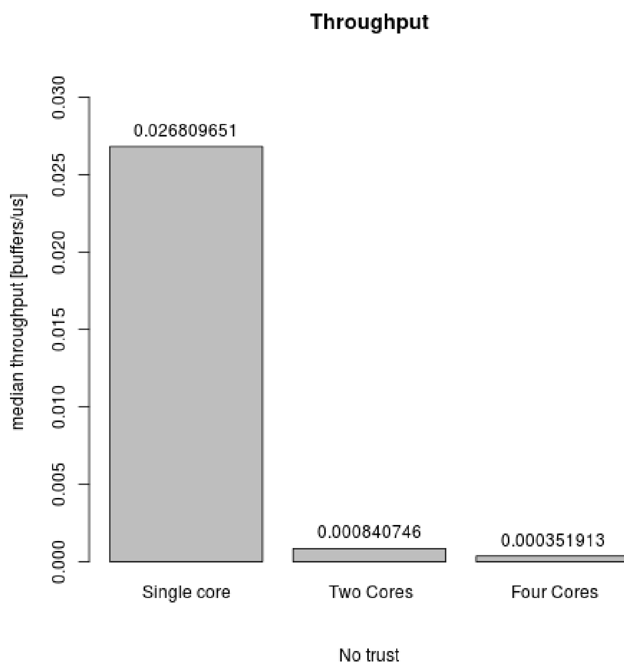

Figure 5.9: Throughput for different core configurations.

5.2.4 Implications For Users

As the differences in both latency and throughput are vastly different for the different configurations, and different still for other bulk transfer backends, users who care about the performance of their application might need to adapt their system to the setup it is running on. This is similar to the way different flounder backends have different performance characteristics – which is also one of the main reasons for the different characteristics of our system. The performance penalty of untrusted cross-core channels is so big, that it might be better to run both domains on the same core even if other cores are idle. In some cases, users may even be willing to pay that performance penalty, as it buys them isolated buffers of arbitrary size.

5.2.5 Possible Improvements

Due to time constraints, we could not try out as many optimizations as we would have liked. One area where there might still be room for improvement is memory allocation, as our backend often allocates and frees small structs of just a few different constant sizes.

However, from our measurements it is clear that for most setups, the performance is completely dominated by the underlying flounder channel. LMP always performs a system call with a context switch to send a message, and in the untrusted cases the revocation alone will take orders of magnitude longer than our local userspace processing. The only setup without any mandatory system calls is a trusted channel over UMP, and it would also proportionally benefit the most from even small optimizations as it is already the case with the most throughput.

Chapter 6

Future Work

It would be interesting to have a working implementation of the shared_frame capability that we described in section 3.5, though for it to work properly would also require the reactivation of the remote capability database in the monitors. The concept of encoding state machines in capabilities could be taken much further, and the limits of the achievable semantics are probably worth exploring.

We were surprised by the decay in throughput as we added more trusted channels over UMP, and the rather extreme result might be worth further research. We suspect the problem to lie with the exploitation of cache-coherency protocols that constitutes the main optimization of UMP. If that really is the case, it suggests that even using cache-coherency exclusively for optimization might not scale well.

It would be interesting to have other bulk transfer backends with the same interface on shared memory machines, to compare to our implementation. Obvious ideas are DMA engines and naive full-copy channels. While these would not be faster than our implementation on the machines we used, they may become more favourable on NUMA machines, as the receiver would get the buffer on its local node.

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] 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.

[6] 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.

[7] Mark Nevill. An evaluation of capabilities for a multikernel. Master's thesis, ETH Zurich, Systems Group, Department of Computer Science, Universitätsstr 6, 8092 Zurich, May 2012.

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

[9] 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`.

[10] 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.

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