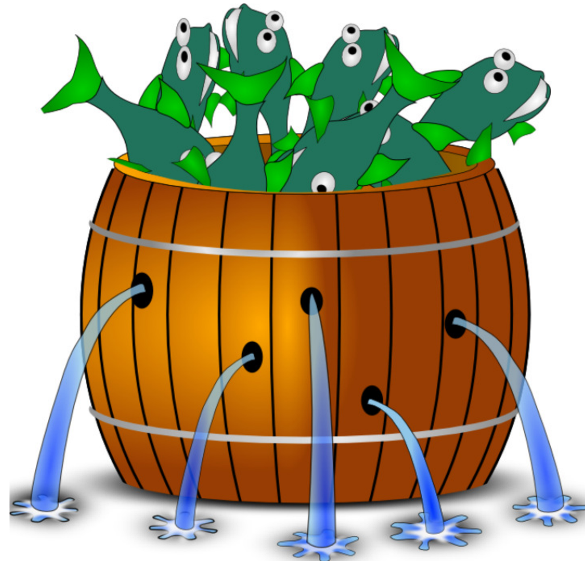
Barrelfish Project
ETH Zurich



Bulk Transfer

Barrelfish Technical Note 014

Pravin Shinde

11.08.2011

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
1.0	11.08.2011	TR	Initial version

Contents

1 Introduction 5
- 1.1 Existing implementation in Barrelfish 5
  - 1.1.1 $bulk_transfer.c/h$ 5
  - 1.1.2 pbufs 6
- 1.2 Related work 7
- 1.3 Requirements of Barrelfish 7

2 Design 8
- 2.1 Bird's eye view of design 8
- 2.2 Terminology used 8
- 2.3 The Producer 9
  - 2.3.1 Generator 9
  - 2.3.2 The Producer abstraction 9
  - 2.3.3 Shared-pool 10
  - 2.3.4 A meta-slot structure 10
  - 2.3.5 shared and private data-structure 11
- 2.4 Consumer 11
  - 2.4.1 Consumer-queue 11
  - 2.4.2 shared and private data-structure 13
- 2.5 Events/notification/communication between consumer and producer 13
  - 2.5.1 From Producer to Consumer 13
  - 2.5.2 From Consumer to Producer 14
  - 2.5.3 The communication between producer and generator 15
  - 2.5.4 The communication between consumers 15
- 2.6 Initialization 15
  - 2.6.1 Producer initialization 16
  - 2.6.2 Consumer registration 16
- 2.7 Memory management (slot management) 16
  - 2.7.1 Slot state machine 16
  - 2.7.2 Slot management inside producer 17
  - 2.7.3 Slot management inside consumer 17
  - 2.7.4 Freeing up the slots 18
  - 2.7.5 Slot size 18
- 2.8 Security and trust model 18
  - 2.8.1 Slot access security 18
  - 2.8.2 Security against memory invalidation 18
  - 2.8.3 Security against data sniffing 19
- 2.9 Privacy model 19
  - 2.9.1 Relaxed privacy model 19
  - 2.9.2 Strict privacy model with additional data-copy 19
  - 2.9.3 Strict privacy model with early classification support from generator 20
- 2.10 Adaptability with different hardware 20
- 2.11 Limitations 20

2.12 Conclusion 20

Chapter 1

Introduction

This technical note is aimed to describe the design of suitable cross domain bulk transport for Barrelfish. This chapter gives a motivation for this design by providing the details about existing implementations of bulk-transport in Barrelfish and reasoning about why those are not good enough. The chapter 2 contains the actual details about the design. So, if you are already convinced of the need for better bulk-transport in Barrelfish, then you can directly jump to the chapter 2

1.1 Existing implementation in Barrelfish

There are two bulk transport mechanisms that exist in the current(as of August 2011) implementation of Barrelfish. $lib/barrelfish/bulk_transfer.c$ and *pbufs* used in the network stack. We will discuss them briefly in following two sub-sections.

1.1.1 $bulk_transfer.c/h$

This is the official bulk transfer method supported by Barrelfish. In brief, this facility works by sharing a continuous piece of memory (in form of capability) between two domains. These domains then map this physical memory into their virtual address-space. The virtual address where this shared memory is loaded can be different. So, participating processes can't exchange the virtual addresses directly. Bulk transfer mechanism works by dividing the entire memory area into the blocks of fixed size. The domain which initiated the bulk transfer is responsible for managing these blocks. This responsibility includes following.

1. Allocation and deallocation of the blocks.
2. Maintaining the information about free and used blocks.

So, this bulk transfer facility works in master-slave setup. The master allocates the blocks and passes the index of the block to slave. Once the block is passed to the slave, master should not touch it. The slave can locate the block by adding the `(index*block-size)` to the virtual address pointing to the beginning of the shared memory area. Now slave can read/modify the contents in the memory area of this block. Once the slave is done with accessing the block, it can pass the block-index back to the master. Master can either access the block or release it for future use.

Limitations

Security This transfer mechanism can only be used between domains which are co-operative and willing to follow the protocol set for using the bulk transfer mechanism. Malicious domain can corrupt

the data in shared memory by writing at random locations in it or can refuse to return the blocks once passed to it.

More than two domains The design of this bulk transport does not stop you from using it with more than two domains as long as the protocol is followed. But current implementation does not track which domain is holding the buffer. So, one can use the current implementation with multiple domains as long as the domains are co-operative and applications are willing to deal with added complexity of tracking which domains hold which buffers.

Current usage Due to the relatively simple nature of the implementation, its use is limited only to few places. And hence it is not throughly tested in various scenarios.

1.1.2 pbufs

The network stack uses its own custom bulk transport mechanisms. Lets call them pbufs. These pbufs work in similar way to above mechanism, but are more flexible. The application shares some piece of physical memory with network driver which is used as shared memory. Then application creates a list of pbuf structures, each one of them holding an offset within shared memory, length, *pbuf-id* and the shared memory id to which they belong. The key difference here is that the these pbufs may not hold consecutive memory locations even when *pbuf-id*'s are consecutive. In contrast with Barrelfish bulk-transfer where buffer-id value is enough to find the location of buffer within shared memory, pbuf needs to store the offset separately in another list. So, pbufs provide another layer of indirection to allow more flexible use of memory.

The way this mechanism currently works is that, application creates pool of initial pbufs and registers them with network driver. Both, application and network driver maintain the list of pbufs in their private memory. This list is kept in synchronization by sending explicit messages. Now each pbuf can in principle point to any buffer of any size, located anywhere in the shared memory.

This flexibility is used by application when it receives a data from the network driver. Application creates a new pbuf structure with same pbuf-id but pointing at new buffer location and send it back to the driver as new free pbuf to use. And the location of previous buffer is used for processing the data. This way, application can return the pbufs back to driver ASAP without getting affected by how long does the data processing takes. When application is done with processing the data in that buffer, it releases that buffer. This released buffer is then used to create new pbuf which will be registered again with driver in future.

Limitations

Needs more memory The shared memory needs to be bigger than the memory shared with network driver in form of registered pbufs. This is because, at any given point in time, some pbufs will be in application data processing phase and hence can-not be used by driver to receive the new data.

Data corruption This bulk-transfer mechanism assumes co-operative domains. Both domains have read/write access to the shared memory at all the time. This means that if they do not follow the protocol correctly, they may end up writing at same physical location, leading to data corruption.

Complicated memory reclamation The current implementation of pbufs assumes that application can have multiple threads and all of them can access the data in pbufs which are delivered to the application. This complicates the problem of detecting when exactly all threads are done with accessing the particular pbuf. The current implementation uses a reference count mechanism for this detection.

Even though this memory reclamation is functional, it is complicated and one can easily get it wrong, leading to memory leaks.

Supporting more than two domains In theory, this design can be used with more than two domains which are co-operating with each other, but implementation is not designed with such a case in mind. The problematic issues will be tracking which memory buffer is with which domain, and when the memory buffer can be reused.

More issues with current network stack implementation

Following issues are not exactly due to the bulk transfer mechanisms, but are mainly the implementation issues of network stack. I am documenting them here for sake of completeness. Also, knowing the issues in current stack will help in understanding the design decisions made.

Favors the setup with driver on different core It is optimized for the case where application and network driver are running on separate core. It is reasonable to prefer such cases for multicore architectures where presence of large number of cores is assumed. This preference has resulted on overly dependence on efficiency of UMP messages. Current implementation sends around 2 messages (send-packet, tx-done) for transmitting single packet assuming that packet fits into one pbuf. Similarly on packet receiving side, each received packet involves 2 messages (register-pbuf, packet-received). The performance impact of these messages is not strongly visible when application and driver are on different core. But there are cases like SCC where you may end up running the driver on same core as an application. In such scenarios LMP's are used for communication which involves doing context-switch in sending every message. This leads to lot of performance in-efficiencies when current network stack implementation is used in LMP contexts. Specially on those hardware where cost of context-switch is relatively high.

1.2 Related work

The issues of bulk-transport has been investigated quite a few times in the past by work like *F-bufs*, *R-bufs* and *Beltway buffers* so the solution of this problem is known to the large extent. What this document is aiming is to choose right approach and right solution which will suit the requirements of the Barrelfish.

1.3 Requirements of Barrelfish

Following are the ideal requirements from the bulk-transport mechanism.

1. Avoid data copy as much as possible, if can't avoid, then try to push it into user-space/user-core.
2. Ability to batch the 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).

The solution that we are trying to design tried to satisfy as many of these requirements as possible.

Chapter 2

Design

This chapter discusses the design of bulk-transport mechanism that will be implemented in future for Barrelfish network stack.

2.1 Bird's eye view of design

We are aiming to design a cross domain bulk transport mechanism, which supports following features.

1. Should reduce the data copy as much as possible.
2. Should exploit the fact that complete data-isolation is not always needed.
3. More than two separate domains should be able to share the data without copying it.
4. Number of explicit notifications needed should be low.
5. It should work in *single producer, single consumer* and *single producer, multiple consumers*

2.2 Terminology used

This section briefly explains the term used in the design description. This should help in the understanding by reducing the ambiguity of the terms.

1. **Generator:** An entity which is generating the data. It can be software or hardware entity. For example, Network Interface Card(NIC) is a generator, or the application code which is generating the packets to send over the NIC.
2. **data-element:** The data-element is what generator is generating. In case of NIC device, a packet will be a data-element.
3. **Producer:** An entity which is managing the generated data. This management contains providing access mitigation, notifying interested consumers, reclaiming the memory when data-element is consumed. The example of producer will be *NIC device driver*.

   The distinction between generator and producer is rather fine. They are intentionally kept separate because generators can have hardware constraints which does not allow it to provide all functionalities.
4. **Consumer:** An entity which is consuming the data-elements.
5. **Slot:** The contiguous piece of memory where data-element is entirely or partially stored. A data-element can span over more than one slots.

6. **Shared-pool:** A contiguous piece of memory accessible in read/write mode to producer and accessible in read only mode to consumers. Shared pools are divided into above slots.
7. **Production-pool:** It is a collection of all shared-pools used by the particular producer. The initial shared-pool is added by the producer, and later on every consumer which joins the system contributes a new shared-pool to the production-pool.
8. **Free slot:** A slot which is available, and can be given to generator.
9. **In-generation slot:** A slot is given to generator. Only the generator should read/write this slot now. This is the only mode when data can be written in the slot.
10. **In-consumption slot:** One or more consumers are currently consuming this slot. In this mode, slot is strictly read-only no one should modify the data in slot in this mode.
11. **Slot-pointer:** A data-structure which holds enough information to locate particular slot present in any shared-pool with producer. These are typically used in cross domain communication to identify the exact slot. A typical slot-pointer should have a shared-pool-id and offset within shared-pool to find the particular slot.
12. **Classification:** This term refers to the classification of data-elements between consumers. In other words, it is a process of deciding which consumers should receive this particular data-element. In network stack parlance, this classification will map to deciding which process should get the received packet.

2.3 The Producer

This section describes the producer and generator together with their responsibilities. Even though producer and generator can be a different entities internally, the consumers will only see the producer interface.

2.3.1 Generator

Lets make a rough sketch of typical generator. Here we are aiming for the NIC devices. Typically, these devices will have an internal queue (RX-queue) of slots where the packet received from wire will be copied. These generators are also capable of generating a notification in form of interrupt. Now the way RX-queues work on different devices differ in some way. Few devices expect contiguous memory in the slots whereas other devices are capable of DMAing the packet in non-contiguous memory as well.

The software generators like application logic which creates new packet to send out is much simpler than NIC devices but can have quirky behavior like above described NIC devices.

From bulk-transport point-of-view we will be treating the generator as a part of producer.

2.3.2 The Producer abstraction

The producer abstraction is nothing but the combination of generator and remaining producer functionalities. The producer abstraction is what consumer sees and interacts with. It is responsible for managing the data produced by generator and using shared-pools to get that data till consumers. It is also responsible for managing the shared-pool memory and co-ordinating the access to this shared resource.

2.3.3 Shared-pool

Shared-pool is the area where producer will generate the data and consumers will read it from. The producer breaks this shared-pool into *slots*. The size of slot is based on the capability of generator and should be multiple of cache-line size. The details of how to choose the slot size and the side-effects of small or large size are discussed in subsection 2.7.5.

The producer is responsible for managing these shared-pools, and it does by maintaining the list of shared-pools. At the time of producer initialization there will be only one shared-pool which is added by the producer itself. As new consumers join the system, each one of them will contribute an additional shared-pool.

Releasing the shared-pool

If any consumer decides to leave the system, then producer stops using the shared-pool provided by him. The producer will wait till all the slots in that shared-pool are free and then it will ask all consumers who have mapped this shared-pool to release the pool. Once all consumers release the pool then producer can also release the pool and inform the consumer who contributed the shared-pool about completion of the process.

This is rather long process and depends on all the consumers for it's completion. But this process is not part of the critical path, it is only part of the tear-down process so some inefficiency can be tolerated here.

Ideal size of production-pool

The production-pool is just a collection of all valid shared-pools available at given time. Ideally, production-pool should be big enough to accommodate the queue of the generator (ie. All the descriptors in RX-queue of NIC device) and all the data-elements which are with consumers and are not released back (in-consumption state). So, the total memory size should be proportional to the capacity of the generator(ie. NIC device) and number of consumers. So, one way to look at this is that, producer will provide shared-pool which can satisfy the needs of generator and each consumer will contribute a shared-pool to allow in-consumption data-elements. Each consumer is allowed to keep only fixed amount of in-consumption data-elements based on the size of shared-pool it contributed. This way, fixing the size of allowed pending in-consumption data-elements limits the ability to consumers to over-consume the slots from production-pool.

2.3.4 A meta-slot structure

This structure hold the additional information about each slot in a shared-pool. It can be seen as index table on shared-pool. This structure is private to producer and is used to manage the slots within shared-pool. Following are the key elements of this structure.

1. **slot-id:** The slot identifier.
2. **offset:** The location of slot within the shared pool.
3. **data-len:** Size of valid data in the slot.
4. **state:** The state of slot (Free, in-Generation, in-Consumption)
5. **consumer-list:** Which consumers are accessing it?
6. **Next, prev:** Used to maintain the list (eg. Free slots)

The producer maintains a separate meta-slot list for very shared-pool available with the producer. This setup enables producer to track every slot and its state.

Other than this, the producer also maintains the list of consumers who have registered.

2.3.5 shared and private data-structure

This section briefly describes which of above data-structures are private and which of them are shared with other entities.

Private data-structures

The producer maintains lot of state in private data-structures. It includes following.

1. List of consumers connected. This list also maintains the state of each consumer with respect to the communication with producer.
2. List of shared-pool-meta-data. This meta-data includes following information.
   (a) Consumers having access to this shared-pool.
   (b) meta-slot structure containing private information about state of each slot in this shared-pool.
   (c) List of free slots in this shared-pool.
   (d) The generator-queue which is used by generator to create data-elements is shared with generator. But as we treat generator as internal entity, we mark this as private data instead of shared data.

Shared data-structures

The producer shares following data-structures with different entities.

1. Shared-pools can be shared with any consumer.
2. For every consumer, a consumer-queue (see 2.4.1) is pairwise shared with that consumer.

2.4 Consumer

This section describes the internals of the consumer. Consumer mainly consist of a consumer-queue data-structure which allows sharing of slots between producer and consumer. Consumer also has read-only access to the shared-pools.

2.4.1 Consumer-queue

This section describes the consumer-queue in details. The consumer-queue is a shared data-structure between consumer and producer and it is exclusive between each pair of consumer-producer. This data-structure is also maintained on contiguous shared memory which is read/writable by both consumer and producer. This data-structure is shared ring-buffer with few virtual-registers which are used to manage the access the ring-buffer.

1. **write-register:** This virtual register is the first element in the consumer-queue. This element should be of size cache-line to avoid any cache-conflicts. This element can be modified only by the producer and consumer can only read it. The value in this virtual-register contains the write-index for consumer-queue. *The write-index points to the slot-pointer within consumer-queue that producer will provide next.*

2. **read-register:** This virtual register is the second element in the consumer-queue. This element should be of size cache-line to avoid any cache-conflicts. This element can be modified only by the consumer and producer can only read it. The value in this virtual-register contains the read-index for consumer-queue. *The read-index points to the slot-pointer within consumer-queue that consumer will consume next.*

3. **queue-size-register:** This virtual register is the third element in the consumer-queue. This element can be modified by producer and consumer can only read it. The value in this virtual-register indicate the number of slot-pointers which are valid in the consumer-queue. This register allows dynamic adjustment queue-size based on the current load. Similar feature is implemented in the *beltway buffers* with the claim that keeping the size of queue as small as possible helps in cache friendliness. This feature is actually optional and will be added in later stages. To implement this feature, one also need to made decision like when the queue size should be reduced and when it should be increased again. Producer can make these decisions based on how much preference it wants to give to this particular consumer over others.

4. **Slot-pointers:** After above three registers, the rest of the space in consumer-queue is used for storing slot-pointers. The slot pointers are used to point a particular slot in one of the shared-pools within production-pool of the producer. Slot-pointers have following information.

   (a) `shared-pool-id:` Id of the shared-pool which is holding this particular slot. These id's are given by the producer and will be unique within that particular producer for given shared-pool. As the shared-pools within particular producer may increase/decrease over time, the consumer may receive a slot-pointer with shared-pool-id which it has not mapped yet. In such a case, it should send a message to producer asking for read-only access to this new shared-pool and then map it into the virtual address-space. Once the shared memory frame associated with shared-pool memory is mapped, consumer can continue to access the slot using slot-pointer.

   (b) `slot-index:` The index of slot within that shared-buffer. This value is only useful for producer as this slot-index maps into the meta-slot structure which is private to the producer. Consumer should not alter this value. In case malicious consumer alters this value, the producer will be able to detect it as meta-slot structure maintains the information about which all consumers are currently consuming the slot and whenever the producer reclaims the slots freed by consumer, it validates if this consumer was consuming the slot being released. The reason for maintaining this information in slot-index even when it is not useful to consumer is because it speeds up the producer in relocating the slot within shared-pool when it is freed by the consumer.

   (c) `Offset:` The start of the slot within shared-buffer. In case of fixed size slot, offset can be calculated by slot-index.

   (d) `More:` As the data-element can span more than one slots, this flag tells us if there are more slot-pointers following which belongs to same data-element. This flag is equivalent to the **More Fragments** bit in the **IP protocol**. In contrast to IP protocol there is no fragment identification number, but the slot-pointers belonging to same data-elements are assumed to follow each other. So, the order in which the slot-pointers are added to the consumer-queue is important. As we have only one producer which is adding the slot-pointers in *available to consume* section and also this producer is dealing with one data-element at one time, maintaining this order is fairly simple.

Conditions on consumer-queue registers

1. Queue empty condition: read-index == write-index
2. Queue full condition: ((write-index + 1) % size) == read-index
3. Elements available to consume
   (assuming queue not empty):
   if (write-index > read-index)
   then {[read-index, (write-index - 1)]}
   else {[read-index, (size - 1)], [0, (write-index - 1)]}
4. Elements which are already consumed and are now free.
   (assuming queue not empty):
   if (write-index > read-index)
   then {[write-index, (size - 1)], [0, (read-index - 1)]}
   else {[write-index, (read-index - 1)]}

2.4.2 shared and private data-structure

This section briefly describes which of above data-structures are private and which of them are shared with other entities.

Private data-structures

The consumer does not have to maintain lot of private state based on which model of slot-consumption is used (refer 2.7). When doing out-of-order consumption it will have to maintain some state about which slots are in consumption and which slots are free and ready to go back to producer. Every consumer will also need some private state to remember the state of the producer and the consumer-queue status. If this consumer is directly communicating with other consumers then it will also need to maintain some private state about that communication (refer 3).

Shared data-structures

The consumer shares following data-structures with producer and other consumers.

1. The shared-pool that it has contributed to producer and potentially shared with all consumers.
2. The consumer-queue is exclusively shared with producer.

2.5 Events/notification/communication between consumer and producer

This section describes the communication between the consumer and the producer.

2.5.1 From Producer to Consumer

This paragraph describes the events sent by producer to consumers.

1. **More data arrived:** This notification is sent to consumer whenever the consumer-queue is empty and new data-element arrived. This callback is not triggered for every arrival of data, but only when consumer is not explicitly polling for data. Whenever new data arrives on the empty queue, producer can assume that consumer is not polling the channel as queue is empty, and hence producer should send this notification to consumer to wake it up. In other case where queue is non-empty, consumer is already aware of the presence of data-elements there. Consumers are expected to deal with dynamically growing of queue-size and hence adding more elements to non-empty queue without sending explicit notifications should not break the consumer-logic.

2. **Consumer-queue full:** This notification is sent to consumer whenever consumer-queue is full and producer is not able to add new data-elements to the consumer-queue. Triggering of this notification means that consumer is slow in consuming the data from the consumer-queue. And result of this producer is going to drop the data-elements which were aimed for this consumer till there is more free space with this consumer. This message also means that producer is not going to actively check if consumer-queue has a free space or not. It is a responsibility of consumer to send a notification to the producer whenever it is ready to receive more data.

3. **Consumer-queue almost full:** This is an optional notification and is designed as refinement of the basic approach. This additional event can be called when the number of free slot-pointers in the queue drops bellow certain threshold. This notification can be used as a warning sign to the consumer that either it takes some action to free up more slot-pointers in consumer-queue, or producer will soon start dropping the data-elements addressed to this consumer.

4. **Error event:** This notification is sent when something unexpectedly goes wrong. The information and severity of the error will inform the consumer that if the problem is transient or permanent. And based on the problem type, consumer can take corrective actions. Few samples of these errors are corruption of consumer-queue, fetal error in producer, etc.

5. **Add shared-pool:** This is an optional notification which allow producer to push the notifications about newly added shared-pools. This message is optional because consumers can lazily ask for these new shared-pools whenever they encounter them as part of slot-pointer while consuming data-elements. This notification is not considered as part of critical path or part of the data-flow. This notification will be generated only on the arrival of new consumer or if producer decides to increase the available shared-pools. This can be classified as setup or maintenance path.

6. **Remove shared-pool:** This notification is opposite of above *add shared-pool* call. It tells consumer is that producer will not be using a particular shared-pool from now onwards for whatever reason (eg. the consumer which gave that shared-pool frame has terminated the connection with producer). So, every consumer which receives this message should remove the mapping for shared-pool from it's virtual memory. This notification is also relatively rare and will be generated only when one of the consumer decides to quit or if producer decides to reduce the number of available shared-pool. So, this notification should not be considered as part of critical-path or data-flow. It can be classified into setup or maintenance path.

2.5.2 From Consumer to Producer

This paragraph describes the notifications sent by consumer to the producer.

1. **Consumer-queue space available:** This notification is sent to the producer when consumer-queue is full and new free slot-pointer is added creating new space. Once the queue is full, producer is not expected to check the queue status until it receives this message saying that now application has more space to receive packet. As an optimization, consumer might wait till some more free-space is accumulated before informing the producer to re-start the flow. This way the inefficiency similar to *silly window behavior* in TCP flow control can be avoided.

2. **Get frame for shared-pool-id:** This notification is sent to the producer when consumer receives a slot-pointer with shared-pool-id which is not mapped by this consumer yet. By sending this

notification, consumer is asking for the read-only access to this shared-pool. The producer should respond to this message with either with valid capability or with error.

3. **Forwarding slot-pointer to other consumer:** This message is related to the ability of consumers to forward slot-pointers within each other. There are two cases here. In first case, the consumer forwards the slot-pointer to other consumer and then also collects it back before declaring it as free. In this case, producer does not need to be informed at-all about this sharing as consumer is taking responsibility of properly freeing the slot. But in case where consumer just wants to forward the slot-pointer and continue on it's own working without bothering about when and how other consumer is going to stop accessing that slot, then it should inform the producer that it has forwarded that slot to some other consumer. It can be done with this message. The reason we need this separate message is that, because of sharing there will be two consumers which will be releasing the slots and producer should be aware of them. The information from this message is used to update the meta-slot structure inside producer for given slot with one more consumer. This message contains the slot identification information and consumer identification information. The other consumer can now release the slot whenever it is done using it's consumer-queue, and producer can validate and account that operation correctly.

   This forwarding has some more implications which are not fully explored in this document. Here are few leads on that. Firstly, this path allows consumer to receive the slots without being initiated by producer and this slot will not be a part of consumer-queue. So, the producer need to somehow account for these slots so that it can maintain the fairness in the number of outstanding slots per consumer. This mechanism also needs some way by which two consumers can talk with each other and exchange the information like consumer-id's, producer-id and slot-pointers. Also, consumers should be careful in dealing this information as consumer-id and slot-pointers are only valid in context of particular producer.

2.5.3 The communication between producer and generator

The communication between producer and generator is very hardware dependent, and I am not sure if it should be part of this document. Following are the generic notifications which are exchanged between the producer and generator. These notifications are aimed to maintain the generator-queue.

1. Add empty slot (from producer to generator): This message will add a new free slot into generator-queue.

2. Data-element generated (from generator to producer): This message informs the producer that there is new data-element in the generator queue that should be handled. It is equivalent of *packet received interrupt* from NIC hardware to the device driver.

3. Generator-queue full (from generator to producer): This message tells the producer that there is no space left for generator to create more data-elements.

2.5.4 The communication between consumers

TBD: The communication between consumers is needed to share the slot-pointers between them.

2.6 Initialization

This section briefly provides the steps involved in initialization of producer and consumers.

2.6.1 Producer initialization

At producer initialization, following steps will happen.

1. Create and initialize the empty list of shared-pools and meta-slots.
2. Create and add the initial shared-pool big enough to satisfy generator needs.
3. Divide the shared-pool into slots, create a meta-slots list for this shared-pool and add it to meta-slots list.
4. Create an empty list of consumers.
5. Initialize the generator (if needed). This might involve populating the generator-queue with free slots so that generator can start producing data.
6. Loop: wait for next event/message and process it accordingly.

2.6.2 Consumer registration

As part of initialization process, consumer should register itself with the producer. This initialization process includes following steps.

1. Create a new contiguous memory frame and provide it to producer as new shared-pool frame.
2. Get the list of registered shared-pools from producer and map it as read-only in the virtual address-space of consumer.
3. Create a new contiguous memory block to maintain a consumer-queue.
4. Initialize this memory block with consumer-queue data-structure.
5. Share this memory block with producer.
6. Send a notification *consumer-queue space available* to the producer so that it will start sending data-elements.
7. Loop: wait for next event/message and process it accordingly.

2.7 Memory management (slot management)

The slots are most important aspect of this bulk transport mechanism. They hold the data which will be sent across the domains. Now lets see how exactly these slots work.

2.7.1 Slot state machine

Slots are created by the producer in the memory provided by shared-pools. Once created, slot remains valid until and unless the shared-pool holding the slot needs to be freed. In the lifetime of the slot, it goes through following states.

1. **Free:** This is the initial state, in this state slot is free and ready to be used. The producer maintains a list of free slots so that it can quickly find them whenever they are needed.
2. **In-generation:** This is the second stage of slot. In this state, the slot is owned by the generator and no one should access it(not even in read state!). This is the only state in which the data will be written into the slot.

3. **In-classification:** This is the third and transient state. This is the state when generator returns the slot after it has generated the data. The producer will use the classification to figure out which consumers this data-element should go. The producer is free to use any type of classification that is needed. The description of how that classification should happen is not in the scope of this document. Once classified, the producer will update the consumer-queues of each selected consumer with slot-pointer and if needed, send them the notifications. Once the slot-pointers are added into consumer-queue, the state of slot is updated to the **in-consumption**.

   If no consumer is interested, then the slot is marked as free and added back to the list of free slots.

   This is the stage where **zero copy data access** is achieved. As instead of actually copying the data from the slot, slot-pointers are provided to the consumers. These slot-pointers provide zero-copy access and data-sharing capabilities.

4. **In-consumption:** This is the state in which data is available to the consumers. Here, consumer has multiple options about what to do with this slot.

   - **In-order consumption:** Consume the slot as quickly as possible, and once consumed add it to freed slot by moving the read-index. As consumer is dealing with one data-element at a time, there is no need for complex state management and free-slot management inside consumer-queue is sufficient enough without needing any additional state machinery.

   - **Out-of-order consumption:** Saves the newly arrived slot-pointer and replace it with one of the other slot-pointers which has been previously consumed and then add this replaced slot to freed slot by moving the read index. This method allows you to consume the slots at leisure and you can have more than one data-elements in consumption at any given moment of time. It also allows you to free up the slots which don't hold useful data quickly and release the slots with important data later on. This method gives much more flexibility but at cost of added free-slot management within consumer. As consumer is releasing the slot in out-of-order of their consumption, it has to maintain some state about which slots are released and which are still in consumption.

   - **Private-mutable consumption:** If the application needs a private access to the slots where it can modify it then it should make copy of this slot in it's private memory and release the actual slot. This way, application is free to do anything it wants with the private slot. This solution scales well with large number of private-mutable consumers because the data-copy operation is performed by consumers and not by producer. As data-copy operation is typically most expensive, we want to push it out from producer into consumers. If consumers are running on separate cores then the data-copy operation of one consumer will not affect other consumers.

   Once the consumer is done with consumption, the slot will be returned to the producer. And when all consumers who are sharing the slot report that they are done with consumption then producer can mark the slot as free and add it to the list of free slots for future reuse.

2.7.2 Slot management inside producer

The producer maintains a private meta-slot data-structure for each slot to track these slots. It also maintains the list of free, in-generation and in consumption slots.

2.7.3 Slot management inside consumer

The slot management is not needed in consumer if they are following in-order or private-mutable consumption. In these two cases, the consumer-queue implicitly does the slot management on behalf of consumers.

In case of out-of-order consumption, the consumer will have to maintain the list of all slot-pointers which are currently accessible and also the list of slot-pointers which are done with the use and can be released as free slots.

2.7.4 Freeing up the slots

Whenever consumer is done with accessing/consuming the data-element, it should be added back to the free-slot-pointers area of the consumer queue. It is important to note that the order in which slots will be freed by consumer need not match the order in which it consumed the slots. This flexibility allows consumer to take longer time on certain slots while quickly returning the slots which have arrived afterwards. Consumer can hold back certain number of slots without releasing them, but this number depends on the length of the consumer queue. As the producer controls the consumer-queue length it can give some consumers more slots than other based on the configurable policies. The policy that initial implementation of this buffer will be using is that, the size of consumer-queue will be directly proportional to the size of shared-pool that particular consumer has contributed.

2.7.5 Slot size

The size of the slot is greatly dictated by the capabilities of the generator. If generator needs continuous memory to produce data-element then the size of slot will be the size of biggest data-element. If the generator can DMA the data-element into multiple non-contiguous memory locations then, the slot size should be based on the average data-element size. In any case, the slot size must alway be a multiple of cache size and every slot should be cache aligned. This is mostly due to the fact that each data element might get consumed by different consumer running on different cores, and we do not want cache conflicts when they are accessing different data-elements.

The dis-advantage of having slots of size of largest data-element is that it leads to internal fragmentation and waste of memory when packets are small. On other hand, when slot size is small, we reduce the capacity of NIC hardware as typically RX-queue will have limited number of entries, and by using small sized slots, we will fill up the entries quickly.

2.8 Security and trust model

This section discusses the trust model assumed by this bulk transport design and it's security implications.

2.8.1 Slot access security

This bulk transport mechanism works by sharing the memory and it needs security against invalid memory accesses. This security is provided by the slot-pointers. Slot-pointers are designed to be valid across the virtual address-spaces of all consumers. Slot-pointers contain information like shared-pool-id which can be used to find the starting address and length of the pool. When consumer is calculating actual pointer, it can always validate that the pointer lies within shared-pool by making sure that offset is always smaller than shared-pool size.

2.8.2 Security against memory invalidation

The more problematic case is when one of the consumer dies or invalidate the memory given to producer without informing first. This case can lead to invalid memory access in producer, generator and other consumers as one of the shared-pool is not accessible any longer. The protection against such

a case should be provided by operating system by not removing the memory frame as long as it is mapped by at-least one domain/process. If operating system does not provide such an assurance and if the consumers are non-trustworthy then producer should somehow take the complete ownership of the memory provided by the consumer, or provide all the memory itself.

From above discussion, it can be seen that the trust mainly lies with producer and not the consumer. Consumers can't do much damage to producer as all the actions of consumers can be validated by the producer. For example, when any consumer tries to free up the slot, the producer can verify if the consumer really holds that slot or not.

2.8.3 Security against data sniffing

The only damage that consumer can do is to try and read the data-elements which are not destined to it. Consumers can do this as they have read-only access to entire production-pool. Performing this type of snooping is not trivial as all the meta-data about the slots is maintained in the private memory by the producer, so consumer can only guess about location and type of data inside the other visible slots. But nevertheless unauthorized data-sniffing between consumers is theoretically possible in this design.

This shortcoming can be overcome by sharing the shared-pool pairwise between consumer and producer and then by either adding the overhead of performing data-copy, or by using generator capability to perform early classification of data elements. The 2.9 will give more details about these stricter privacy models.

2.9 Privacy model

This bulk transport is designed to work in relaxed privacy model with possibility of tightening the privacy model by either using early classification capabilities of generator or at added cost of data-copy. This section describes all of these modes.

2.9.1 Relaxed privacy model

The bulk-transport design described most of the above document is based the relaxed privacy model. The reason for describing this model in detail is that other models which gives better privacy (described in 2.9.2, 2.9.3) are sub-set of this model hence they are easy to understand once this model is clear. In this model, all consumers have read-only access to entire production-pool. And hence in theory malicious consumers can sniff the data-elements addressed to other consumers. The 2.8.3 gives some description of this issue from the security perspective.

2.9.2 Strict privacy model with additional data-copy

In this model, shared-pools are not shared with all consumers, but every shared-pool is pairwise shared between producer and with only one consumer (one who has contributed that shared-pool). The initial shared-pool contributed by the producer itself is shared between producer and generator. This shared-pool will not be directly accessible to any consumer. All the data generated in this pool which is private to producer and generator and then based on the classification of the data-element, it is explicitly copied into the shared-pool of the selected consumers. This way, other consumers can never see the data-elements which are not destined to them. This stricter privacy policy comes with the cost of additional data-copy. Also, as producer is performing this data-copy, this can adversely affect the throughput of the whole system.

2.9.3 Strict privacy model with early classification support from generator

In this model as well, shared-pools are not shared with all consumers, but every shared-pool is shared between producer and with only one consumer. This model depends on the capabilities of generator to provide stricter privacy model. If the generator is smart enough and can classify the data-elements before they are created, then these data-elements can be directly created into the shared-pool of selected consumer. This avoids additional copy from the producers pool to the shared-pool of the consumer.

This is an ideal model which provides strict privacy without compromising on the performance. But it needs smarter generators. The recent NIC hardwares like **Solarflair** and **e10000** are capable of performing such a classification.

2.10 Adaptability with different hardware

This section discusses the ability of this design to adapt with different designs and features of NIC hardwares. We believe that the interface provided by the producer is generic enough to cover most of the functionalities provided by the smarter generators. And as we can control the integration between producer and generator without affecting the consumers, we can easily adapt to exploit the different hardware features provided by the generators. One of the example of such an adaptation can be seen in 2.9.3, the design discussed here easily adapted to exploit the early classification capabilities of the generator to give stricter privacy based zero copy bulk transport.

2.11 Limitations

This design still misses some good features in favor of keeping the complexity reasonable. These features include support for multiple-producer, multiple-consumer setup and ability for consumer to specify where data-element should be generated (true zero copy receive).

2.12 Conclusion

This chapter provides the details about the new cross-domain transfer facility design. This facility allows zero-copy network implementation which can support sharing between more than two domains as well as multi-level sharing (sharing by consumers between consumers). This solution also reduce the number of messages exchanged by sending them only when it is absolutely necessary.