|  |  |
| --- | --- |
|  | 学校代码： 10246 |
|  | 学 号： 17210240136 |
|  |  |



|  |
| --- |
| 硕 士 学 位 论 文 |

**（专业学位）**

|  |
| --- |
| **基于NTB的传输协议的设计与实现** |

|  |
| --- |
| **Design and Implementation of Transmission**  **Protocol Based on NTB** |

院 系： 计算机科学技术学院

专业学位类别（领域）： 计算机技术

姓 名： 李璟岓

指 导 教 师： 梁 瑾 高级工程师

完 成 日 期： 2019年 9月 14日

**指导小组成员名单**

张世永 教授

钟亦平 教授

吴 杰 教授

梁 瑾 高级工程师

吕智慧 副教授

吴承荣 副教授

[摘要 5](#_Toc20610499)

[Abstract 6](#_Toc20610500)

[第一章 绪论 7](#_Toc20610501)

[1.1问题的提出 7](#_Toc20610502)

[1.2 研究的背景和意义 8](#_Toc20610503)

[1.3 论文的主要工作 10](#_Toc20610504)

[1.4 论文组织结构 11](#_Toc20610505)

[第二章 相关研究及技术概述 12](#_Toc20610506)

[2.1 PCIe总线结构 12](#_Toc20610507)

[2.2 NTB的传输过程 14](#_Toc20610508)

[2.2.1 在BIOS中启用NTB 14](#_Toc20610509)

[2.2.2 地址翻译过程 15](#_Toc20610510)

[2.3 DPDK整体架构 18](#_Toc20610511)

[第三章 数据传输协议NDTP 20](#_Toc20610512)

[3.1 NDTP的主要特点 20](#_Toc20610513)

[3.2 实现NTB的用户态驱动 20](#_Toc20610514)

[3.3 NDTP管理Server连接 24](#_Toc20610515)

[3.3.1 多物理机管理 24](#_Toc20610516)

[3.3.2 多用户连接支持 26](#_Toc20610517)

[3.4 NDTP Socket 27](#_Toc20610518)

[3.4.1 NDTP Socket的发送/接收缓冲区 28](#_Toc20610519)

[3.4.2 Socket缓冲区的无锁环形队列 29](#_Toc20610520)

[3.4.3管理缓冲区内存节点的MemPool 33](#_Toc20610521)

[3.5 NDTP的流量控制 34](#_Toc20610522)

[3.5.1 NTB Sublink流量控制 34](#_Toc20610523)

[3.5.2 NDTP Socket的流量控制 35](#_Toc20610524)

[3.6 NDTP Message结构设计 36](#_Toc20610525)

[3.6.1 NDTP Message MTU 37](#_Toc20610526)

[第四章 NDTP管理进程架构设计 39](#_Toc20610527)

[4.1协议总体架构 39](#_Toc20610528)

[4.2 模块设计及功能说明 40](#_Toc20610529)

[4.2.1 NDTP管理模块 40](#_Toc20610530)

[4.2.2 Shared Memory MGT模块 40](#_Toc20610531)

[4.2.3 NTB MGT模块 41](#_Toc20610532)

[4.3 协议传输过程 43](#_Toc20610533)

[4.3.1 NDTP MGT进程的启动 43](#_Toc20610534)

[4.3.2 NDTP Socket连接的建立 44](#_Toc20610535)

[4.3.3 NDTP Socket连接的释放 44](#_Toc20610536)

[4.3.4 应用程序数据发送过程 45](#_Toc20610537)

[第五章 实验与分析 47](#_Toc20610538)

[5.1 实验环境介绍 47](#_Toc20610539)

[5.2 实验环境配置 47](#_Toc20610540)

[5.3 系统性能测试 48](#_Toc20610541)

[5.3.1 NTB顺序读写性能 48](#_Toc20610542)

[5.3.2 NTB随机写入性能 51](#_Toc20610543)

[5.3.3 NDTP MGT性能测试 52](#_Toc20610544)

[5.3.4 传输时延对比测试 53](#_Toc20610545)

[第六章 总结与展望 54](#_Toc20610546)

[6.1 论文工作总结 54](#_Toc20610547)

[6.2 未来工作展望 54](#_Toc20610548)

[参考文献 56](#_Toc20610549)

[致谢 59](#_Toc20610550)

## 摘要

光纤信道、InfiniBand、以太网等多服务器互连技术在当今高性能计算的设计中起着至关重要的作用。最近，有人提议将PCIe网络作为一种高性能、低功耗的机架内系统互连解决方案来推广使用，用一个PCIe交换机取代一个机架顶部(TOR)以太网交换机，该交换机允许机架中的所有服务器共享机架中的连接的PCIe设备，并通过PCIe链路相互直接通信。所使用的NTB(Non-transparent bridge)技术是PCI-SIG标准的一部分，旨在使两个或多个PCIe域能够像在一个域中一样相互操作。NTB的独特之处在于它能够接受从一个源PCIe域发起的内存读/写操作，并翻译它的目标地址，然后在另一个PCIe域中交付和执行它。然而，NTB是直接通过PCIe总线进行数据传输的，没有提供给用户统一的传输接口，用户进程很难直接使用其进行通讯。

为了解决这一问题，本文对于目前的NTB传输过程进行了梳理，总结出当前NTB技术存在的问题和不足。重点进行了NTB用户态驱动开发、基于NTB的传输协议NDTP架构设计与实现以及传输性能测试。

经过具体的实验表明，NDTP的性能开销在可以接受的范围内，同时与以太网、RDMA等传输方式相比，在时延方面占有一定的优势。

关键字：NTB，PCIe，协议开发，互连

## Abstract

Multi-server interconnect technologies such as Fibre Channel, InfiniBand, and Ethernet play a vital role in today's high-performance computing designs. Recently, it has been proposed to promote PCIe networks as a high-performance, low-power system interconnect solution, replacing a rack-top (TOR) Ethernet switch with a PCIe switch that allows all of the racks. The server shares the connected PCIe devices in the rack and communicates directly with each other through the PCIe link. The NTB (Non-transparent bridge) technology used is part of the PCI-SIG standard and is intended to enable two or more PCIe domains to operate as if they were in one domain. NTB is unique in that it can accept memory read/write operations initiated from a source PCIe domain and translate its target address and then deliver and execute it in another PCIe domain. However, the PCIe bus protocol based on NTB is similar to the IP layer in TCP/IP networks, and it is difficult for user processes to communicate directly with them.

In order to solve this problem, this paper combs and studies the current NTB transmission process, and summarizes the problems and deficiencies of the current NTB technology. The focus is on NTB user-mode driver development, NTB-based transport protocol NDTP architecture design and implementation, and transmission performance testing.

The specific experiments show that the performance overhead of NDTP is within an acceptable range, and it has certain advantages in delay compared with Ethernet, RDMA and other transmission methods.

Keywords：NTB，PCIe，Protocol development，Interconnection

## 绪论

PCI Express (PCIe) [1]最初是作为一种本地总线互连技术设计的，用于连接计算机内的CPU、GPU和I/O设备，后来被增强为一个成熟的交换网络，具有点对点链接、逐跳流控制、端到端重传等功能。非透明连接桥（Non-transparent bridge，NTB）一般为至少连接两个系统的N-N PCI-E总线。配置好后，对设备的PCI映射内存的写操作将被镜像到远程系统上的内存缓冲区。这个“内存窗口”的最大大小是通过BIOS (Intel/AMD)或EEPROM固件或运行时寄存器设置(IDT/Switchtec/PLX)配置的。最近，研究人员希望使用NTB技术进一步将PCIe扩展到机架内主机间互连，旨在连接同一机架内的多个主机[19]。为了有效地将NTB应用于此类用例，需要制定额外的通信协议。

### 1.1问题的提出

光纤信道、InfiniBand、以太网等多服务器互连技术在当今高性能计算的设计中起着至关重要的作用。最近，有人提议将PCIe网络作为一种高性能、低功耗的系统互连解决方案来推广使用[7,8,13,14] ，用一个PCIe交换机取代一个机架顶部(TOR)以太网交换机，该交换机允许机架中的所有服务器共享机架中的连接的PCIe设备[2,3,4,5,6]，并通过PCIe链路相互直接通信[7,8,9]。PCIe总线是一种具有串行点对点全双工通道的交换网络，其中每个附加的PCIe设备都通过由一个或多个通道组成的链路连接到网络。连接到PCIe的设备集形成一个PCIe域，其中一个作为根复合体，根复合体负责枚举和配置域中的所有其他设备，通常连接到CPU。

作为计算机内部的总线协议，PCIe天然具有低延迟、高性能的特征。要将PCIe转换为可以在机架内互相连接的主机间通信的连接系统，必须首先令多个主机能够位于同一个PCIe网络上并彼此直接通信。尽管在2008年发布的多根I/O虚拟化(MRIOV)标准[10]提议在一个PCIe域中支持多个根复合体，但是当前还没有真正的MRIOV交换机或设备的商业实现，也没有迹象表明在不久的将来会有。另一种在多个主机之间 [6,9]直接建立PCIe连接来使用PCIe设备的方法是利用一种特殊的称为非透明连接桥(Non-transparent bridge，NTB)的技术,其目的是使多个系统的某些PCIe设备一个作为PCIe域直接访问资源，而不需要涉及后者的根复杂性。

在没有MRIOV的PCIe交换机的情况下，使用PCIe域隔离和地址转换设备(如NTB)实际上成为连接多个PCIe域的解决方案。D. Riley[15]建议使用NTB为共享I/O设备重定向空间配置寄存器(Configuration Space Registers, CSR)，重定向的CSR请求由管理主机代表计算主机处理，以便控制I/O设备。Cheng-Chun Tu [23]提出了一种基于NTB传输的多主机I/O设备共享寻址模型和主机间通信寻址模型，在节点间提供了基于硬件的远程直接内存访问作为机架内服务器之间的通信原语。Intel在DPDK 19.08中包括了NTB Rawdev驱动器提供了两个单独的主机之间的非透明桥接，使得它们可以相互通信[31]。NTB作为外部设备[17,18,19]在如今变得越来越普遍。

尽管作为PCIe总线协议在事务层被设计成无损的流控制并具有重传机制[11,12]，但它缺乏一个更高层的基于PCIe的传输协议，无法保证一个与传统的系统互联技术（如以太网，InfiniBand和光纤通道）相媲美的高可用性和服务性。

### 1.2 研究的背景和意义

在PCIe体系结构中，在任何时候，每个PCIe域都有一个活动的根复合体组件（Root Complex）[1]。每个服务器都有自己的CPU以及与之相连的根复合体组件，因此，理论上任何两台服务器都不能在同一个PCIe域中共存。

NTB是PCI-SIG标准[4]的一部分，旨在使两个或多个PCIe域能够像在一个域中一样相互操作。NTB的独特之处在于它能够接受从一个源PCIe域发起的内存读/写操作，并翻译它的目标地址，然后在另一个PCIe域中交付和执行它。从启动域的角度来看，内存读/写操作在逻辑上是在其域中本地执行的，尽管在物理上是在远程域中执行的。从概念上讲，NTB就像数据网络中的第3层路由器，它隔离了所连接的PCIe域，因此每个PCIe域只有一个根组件保持不变。一个双端口NTB表示两个PCIe端点，每个端点都属于其邻接的两个PCIe域之一。

NTB使用PCIe总线进行数据传输，但目前还没有一个基于NTB的传输协议被设计并使用，以提供与以太网传输层协议如TCP类似的多用户支持、连接管理、可靠传输、流量控制等功能。

鉴于以上的分析，本文解决的问题主要集中于如何设计并实现一个基于DPDK

环境下的NTB的传输协议NDTP(NTB-based Data Transfer Protocol)。这个问题的研究解决，将会面临如下挑战：

* 在Linux系统中，基于DPDK提供的UIO框架，如何实现一个NTB设备的用户态驱动程序，使之符合相关的接口标准。

现有的NTB设备仅有内核态驱动，且只提供了获得内存窗口、翻译基本地址等简单的功能，目前尚未有一个较为通用的驱动程序出现。因此为了基于NTB构建一个传输协议，必须重新构建驱动并丰富现有的接口。

* 在Linux系统中，如何管理多服务器互连情况下每个服务器所属的NTB设备，使其能被多个用户进程所使用。

PCIe是一种总线传输协议，在同一时刻只能有一个设备通过总线发送消息，因此用户进程通过NTB进行传输是独占式的。我们需要设计这个NDTP协议使其支持多用户的使用场景。

* 在Linux系统中，如何为每个建立的连接分配其使用的NTB设备所映射的地址空间。

NTB所属的内存通过MMIO(Memory-mapped I/O)映射到系统的地址空间中，主机内的每个进程都可以通过NTB所属的虚拟地址访问其内存。为了保证多连接情况下的内存安全，NDTP需要管理NTB拥有的物理内存，所有对NTB内存的访问必须通过NDTP提供的接口，从而保证内存安全。

* 如何为NDTP传输协议建立流量控制、消息结构设计、连接管理等机制。

为了实现多用户支持，仿照其他传输层协议的消息首部设计，我们同样需要为NDTP设计一个连接管理机制。同时为了保证可靠传输，一个流量控制机制也是必须存在的。

* 如何设计用户进程与NDTP管理进程的通讯方式，使用户进程得以使用NDTP协议进行数据传递。

用户进程需要有自己的发送/接收缓冲区，为了避免多次内存拷贝，需要考虑如何设计和管理缓冲区的数据结构，并将缓冲区中的内容共享给NDTP管理进程，使得通过NDTP协议的传输更为高效。

### 1.3 论文的主要工作

本文关注NTB四层协议的设计和实现问题，主要做了如下的重要工作：

1. 实现了NTB的用户态驱动程序。

分析了现有的基于内核的NTB设备驱动功能缺陷以及瓶颈，重新实现了一个基于DPDK的NTB用户态驱动程序，同时使用了一些能够提升NTB传输性能的技术，如Memory Write Combining、Hugepage等。

1. 提出了基于NTB的传输协议NDTP各功能的实现方法。

首先分析了PCIe网络提供的底层网络环境并基于NTB的用户态驱动程序，总结了NDTP需要的功能。随后进行了架构设计和相关的消息包头、数据结构设计等。实现了多用户共享、流量控制、首部解析等功能。

1. 设计了管理NDTP协议的NDTP MGT程序的整体架构。

为对NDTP协议的整体运行情况进行管理，本文设计了NDTP MGT管理程序的整体架构，对连接的建立和释放管理，以及用户程序数据包传输的过程进行了完整的叙述。

1. 测试了NDTP协议的传输性能，分析了性能损耗，并与其他传输方式进行对比，总结了NDTP协议的优势。

测试了NTB的纯硬件性能以及实现了NDTP协议之后的性能，分析了添加NDTP协议层的性能损耗。同时与TCP、RDMA等传输方式进行对比，证明了本文设计的NDTP传输协议与其他传输方式相比在时延方面有一定的优势。

通过以上工作，本文实现了一个基于NTB的传输协议NDTP这一可应用于机架内服务器互连场景的传输协议。

### 1.4 论文组织结构

结合以上讨论，本部分主要描述论文的组织结构：

第一章（本章）主要论述了论文的研究问题，分析并设计NTB的所需的传输协议结构，以及该研究的意义，并且论述了论文的主要工作，从整体上说明了论文的研究情况。

第二章主要描述了与本文研究相关的一些研究的背景，主要包括NTB基于的PCIe协议的基本架构、PCIe协议的路由原理、NTB（非透明连接桥）和TB（透明连接桥）的区别。同时还重点介绍了现有的基于内核的NTB设备驱动所提供的库函数，给出了现有的研究成果、存在的问题以及与本文研究的关系。

第三章实现NTB的用户态驱动，并基于这个驱动设计了基于NTB的传输协议——NDTP，提出了多用户共享、流量控制、消息首部设计的实现机制，以及为了优化传输性能所进行的一些设计取舍。

第四章介绍了NDTP传输协议管理进程的总体架构，各个模块提供的功能，连接的建立和释放过程，以及数据包传输的流程。

第五章针对上述研究内容，设计并完成了基于NTB的传输协议——NDTP原型系统，并设计了一组实验进行了性能测试，并对实验结果进行了评估和分析。

第六章针对相关工作同时结合实验的结果，对整个设计的成果做出了总结，以及对未来工作的展望。

## 第二章 相关研究及技术概述

非透明连接桥(Non-transparent bridge, NTB)是连接两个系统的点对点PCIe总线，在两个子系统之间提供电气隔离。非透明桥在功能上类似于透明桥，只是桥的两边都有自己独立的地址域。桥接器一侧的主机将无法看到桥接器另一侧的完整内存或I/O空间。为了通过非透明桥进行通信，每个NTB端点都有一个(或多个)Memory Window暴露于本地系统。对这些Memory Window的写入被镜像到远程系统上的内存。

**2.1 PCIe总线结构**

一个完整的PCIe体系结构包括应用层、事务层（Transaction Layer）、数据链路层（Data Link Layer）和物理层（Physical Layer）。其中，应用层并不是PCIe Spec所规定的内容，完全由用户根据自己的需求进行设计，另外三层都是PCIe Spec明确规范的，并要求设计者严格遵循的。

事务层负责转换操作系统或固件发出的高级PCIe事务，即，内存，I/O，配置和消息，进入PCIe事务层数据包(TLP)。X86机器承载一个PCIe域，根节点位于北桥，它连接CPU、内存和PCIe网络，并实现事务层协议。PCIe域中的每个PCIe设备都由总线/设备/函数ID惟一标识，并得到一组配置空间寄存器(CSR)，这些寄存器被划分为标准化的部分(如设备/供应商ID、命令、基本地址寄存器或BAR等)和特定于设备的部分。占据PCIe插槽的PCIe设备的功能是一个或多个物理功能(PF)，每个物理功能都像一个逻辑函数。

尽管作为PCIe在事务层被设计成无损的流控制和重传机制[9、19],它不支持任何高级作为PCIe网络年代容错机制,保证服务可用性在一定程度上与传统的系统互连技术,如以太网和InfiniBand。

使PCIe独特的是以太网和IB所没有的两个特性。首先，PCIe允许通过正常的加载/存储指令直接访问远程内存。该特性允许程序无需修改即可使用远程内存。它由具有非透明桥接(NTB)功能的PCIe开关支持。NTB提供通过PCIe交换机连接的主机之间的隔离，同时仍然允许主机之间通信。其次，PCIe允许更短的通信通道。典型的以太网和IB通信信道通过以太网和IB适配器通过PCIe接口连接到北桥。由于适配器和附加协议转换(从PCIe到IB或以太网)被从路径中切断，因此基于pci的通信通道被缩短。

PCI Express也可直接让电脑连接装置。但需要特别说明的是，如果系统具备各自独立的PCI Express域，就无法重新连接成为另一个单独的PCIe域。不过，透过PCI Express的非透明连接桥(Non-Transparent Bridge, NTB)可解决此问题。NTB可连接两组不同的PCI域，同时可将某一PCI域中的特定PCI事务，转译为另一PCI域中相对应的事务，如此可使PCI Express可做为系统之间的通讯介质。NTB若搭配PCI Express连接线或Thunderbolt，将可用于系统组成中，以连接多组子系统，亦可连接实际独立的系统。

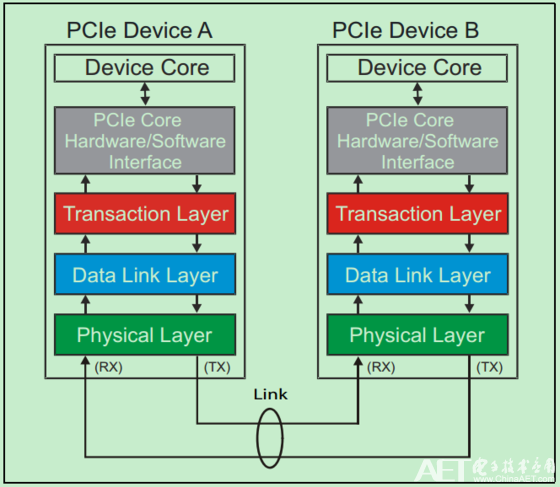


图2.1 PCIe总线体系结构

一个简化的PCIe总线体系结构如图2.1所示，其中Device Core and interface to Transaction Layer就是我们常说的应用层或者软件层。这一层决定了PCIe设备的类型和基础功能，可以由硬件（如FPGA）或者软硬件协同实现。如果该设备为Endpoint，则其最多可拥有8项功能（Function），且每项功能都有一个对应的配置空间（Configuration Space）。如果该设备为Switch，则应用层需要实现包路由（Packet Routing）等相关逻辑。如果该设备为Root，则应用层需要实现虚拟的PCIe总线0（Virtual PCIe Bus 0），并代表整个PCIe总线系统与CPU通信。

**事务层（Transaction Layer）**：接收端的事务层负责事务层包（Transaction Layer Packet，TLP）的解码与校检，发送端的事务层负责TLP的创建。此外，事务层还有QoS（Quality of Service）和流量控制（Flow Control）以及Transaction Ordering等功能。

**数据链路层（Data Link Layer）：**数据链路层负责数据链路层包（Data Link Layer Packet，DLLP）的创建，解码和校检。同时，本层还实现了Ack/Nak的应答机制。

**物理层（Physical Layer）：**物理层负责Ordered-Set Packet的创建于解码。同时负责发送与接收所有类型的包（TLPs、DLLPs和Ordered-Sets）。当前在发送之前，还需要对包进行一些列的处理，如Byte Striping、Scramble（扰码）和Encoder（8b/10b for Gen1&Gen2, 128b/130b for Gen3& Gen4）。对应的，在接收端就需要进行相反的处理。此外，物理层还实现了链路训练（Link Training）和链路初始化（Link Initialization）的功能，这一般是通过链路训练状态机（Link Training and Status State Machine，LTSSM）来完成的。

### 2.2 NTB的传输过程

NTB (Non-Transparent Bridge)本质是一种PCI-Express桥接芯片，它将两台或多台计算机的独立存储系统连接到同一个PCI-Express结构上。由于目前Intel C5500/C3500系列以及之后出产的芯片本身即具备NTB芯片，因此可成为一个普遍使用的低成本、高效能的机架内传输解决方案[18]。

#### 2.2.1 在BIOS中启用NTB

当使用NTB功能时，Intel Xeon处理器C5500/C3500系列仅在1x4或1x8配置中支持端口0上的NTB功能。在单个x16端口配置中不支持NTB功能。BIOS必须启用NTB函数。此外，软件配置enable bit将提供启用或禁用NTB端口的能力。

在这个配置中，两个Intel Xeon处理器C5500/C3500系列UP系统通过每个系统的NTB端口连接在一起，如下图所示。在本例中，每个Intel Xeon处理器C5500/C3500系列系统都支持一个配置为NTB的x4 PCIE端口，而其他三个x4端口则配置为根端口。每个系统都完全独立于自己的PCIe域。

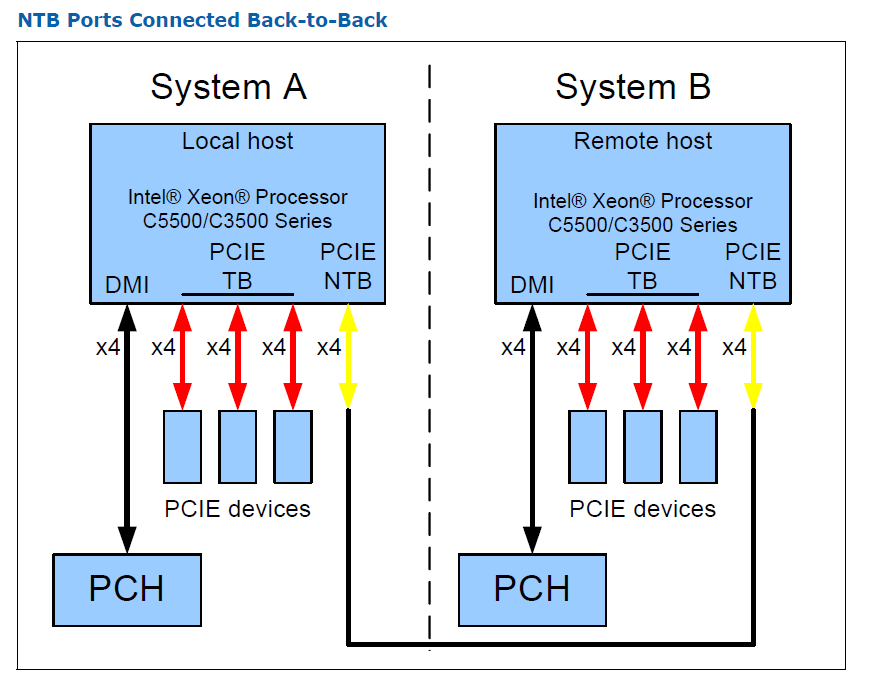


图2.2 NTB端口B2B连接

在上图2.2所示的配置中，一个Intel Xeon处理器C5500/C3500系列(左边的系统)上的NTB端口连接到右边Intel Xeon处理器C5500/C3500系列系统的根端口。第二个系统s NTB端口连接到第一个系统上的根端口，使其成为完全对称的配置。

#### 2.2.2 地址翻译过程

假设我们已经在Bios中预先配置了B2B操作平台和每个方向上的两个内存窗口，之后：

1. 主机A和主机B分别通电(不需要顺序)。
2. 一旦每个系统启动并释放对NTB的控制以进行培训，链接将进入L0状态(link up)。
3. 在每个主机上独立运行的枚举SW将发现并设置与同一主机关联的NTB的主BAR2/3和主BAR4/5寄存器(PB23BASE、PB45BASE)的基本地址指针。此时，只知道内存窗口的大小和位置。例如，4KB到512MB的可预取内存窗口放在一个大小为多个基本地址上。
4. 作为运行时操作期间的最终配置设置，翻译寄存器由与物理NTB关联的本地主机设置，以将事务映射到与接收事务的相应NTB关联的本地系统内存中。这些是SBAR2XLAT和SBAR4XLAT寄存器。

NTB使用指定的Type 0配置头中的栏来定义NTB另一侧的内存空间。NTB支持两组bar，一组位于本地主机接口，另一组位于远程主机接口。每个栏都有可从桥的另一端写入的控制和设置寄存器。地址转换寄存器定义了地址转换方案。极限寄存器用于限制孔径大小。在允许从远程子系统访问之前，必须对这些寄存器进行编程。

Intel Xeon处理器C5500/C3500系列NTB支持入站和出站两个直接地址翻译窗口。这是2/3栏和4/5栏。直接地址翻译用于将一个主机地址空间映射到另一个主机地址空间。NTB是用于连接两个PCIe域的机制，用于转换所有跨PCIe域发送的入站和出站事务。这意味着从NTB的次要端和主要端所遍历的所有事务都被翻译。

将从一个接口转发到另一个接口的地址通过在地址所属的栏内的偏移量中添加一个基本地址来转换，如图2.3所示。

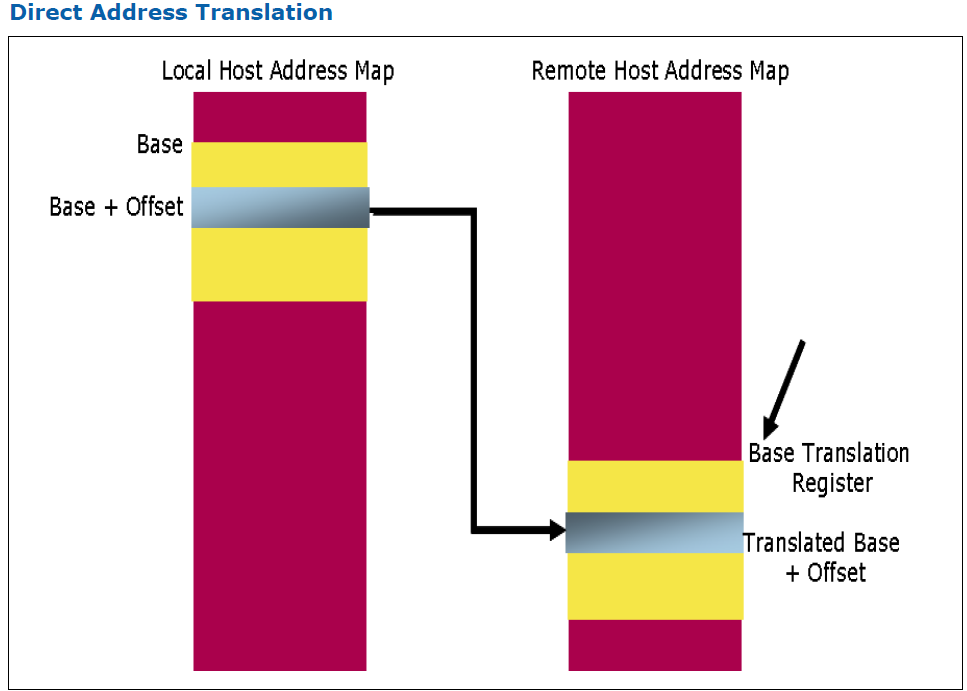


图2.3 直接地址翻译

PCI Express通过3DW和4DW头使用32位和64位地址方案。为了防止地址混叠，所有设备必须解码整个地址范围。本节中的所有讨论都涉及64位寻址。如果使用3DW报头，那么上面的32位地址被假定为0000 0000h。

NTB允许外部PCI Express请求程序通过地址路由的TLPs访问内存空间。PCI Express请求程序可以读写NTB内存映射寄存器或Intel Xeon处理器C5500/C3500系列本地内存空间。入站/出站地址转换过程包括两个步骤：

1. 入站/出站地址检测：

测试看PCI地址是否在为其定义的基寄存器和限制寄存器BAR2/3,4/5内。如果地址位于基寄存器和限制寄存器定义的窗口之外，事务将作为不受支持的请求(UR)终止

1. 地址转换：若为入站，则将远程地址转换为本地物理地址。若为出站，则将本地物理地址转换为远程客户地址。

例如，使用BAR 2/3寄存器将数据包从远程客户地址传输到本地地址映射，从而实现直接地址转换。

地址检测方程:

有效地址=((限制>接收地址>= Base))

寄存器值:

SB23BASE = 0000 003A 0000 0000H——BAR 2/3基本地址，被OS设置为4GB对齐

SBAR2LMT = 0000 003A C000 0000H——将窗口缩小到3GB

接收地址= 0000 003A 00A0 0000H——有效地址转移到平移方程

接收地址= 0000 003A C000 0001H——无效地址，返回一个错误

平移方程:(检测为有效地址后使用)

翻译地址 = (( 接收地址& ~Sign\_Extend(2^SBAR23SZ) | XLAT 寄存器))。

例如，要翻译基于4GB窗口声明的传入地址0000 003A 0000 0000H到一个基于0000 0040 0000 0000H的4GB窗口。

计算过程:

接收地址= 0000 003A 00A0 0000H

SBAR23SZ = 4GB~Sign\_Extend(2^SBAR23SZ) = ~Sign\_Extend(0000 0001 0000 0000H) = ~(FFFF FFFF 0000 0000H) = 0000 0000 FFFF FFFFH)

SBAR2XLAT = 0000 0040 0000 0000H——主侧内存中的基本地址(大小多对齐)

翻译地址= 0000 003A 00A0 0000H & 0000 0000 FFFFH | 0000 0040000 0000H = 0000 004000 A0 0000 h

### 2.3 DPDK整体架构

DPDK( [15],Data Plane Development Kit，数据平面开发套件)是由6WIND,Intel等多家公司开发，主要基于Linux系统运行，用于快速数据包处理的函数库与驱动集合，可以极大提高数据处理性能和吞吐量，提高数据平面应用程序的工作效率。拥有完善的社区，生态形成闭环。早期，主要是传统电信领域3层以下的应用，如华为、中国电信、中国移动都是其早期使用者，交换机、路由器、网关是主要应用场景。但是，随着上层业务的需求以及DPDK的完善，在更高的应用也在逐步出现。

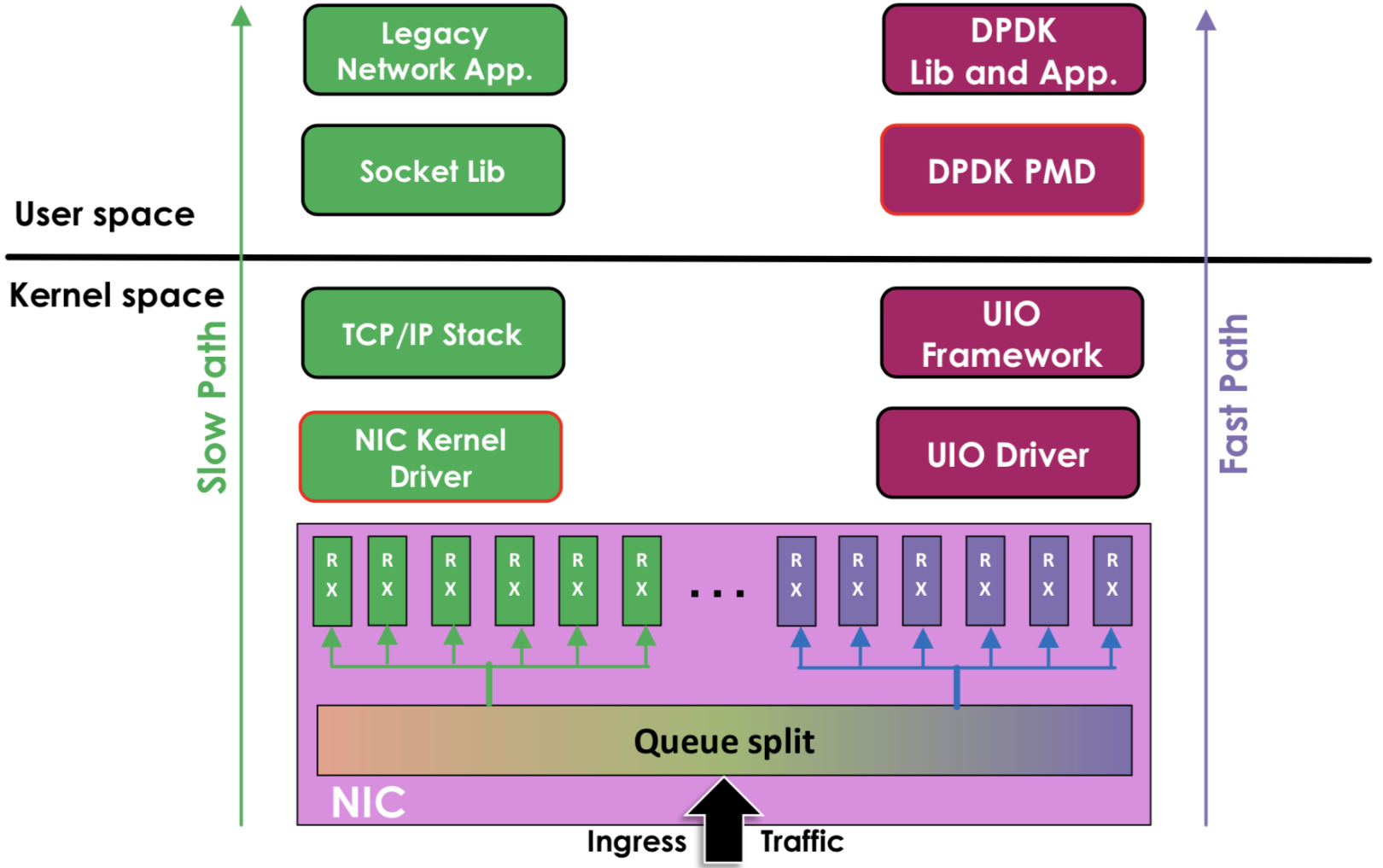


图2.4 DPDK整体架构

左边是原来的方式数据从 网卡 -> 驱动 -> 协议栈 -> Socket接口 -> 业务

右边是DPDK的方式，基于UIO（Userspace I/O）旁路数据。数据从 网卡 -> DPDK轮询模式-> DPDK基础库 -> 业务

用户态的好处是易用开发和维护，灵活性好。并且Crash也不影响内核运行，鲁棒性强。DPDK的UIO驱动屏蔽了硬件发出中断，然后在用户态采用主动轮询的方式，这种模式被称为PMD（Poll Mode Driver）。UIO旁路了内核，主动轮询去掉硬中断，DPDK从而可以在用户态做收发包处理。带来Zero Copy、无系统调用的好处，同步处理减少上下文切换带来的Cache Miss。运行在PMD的Core会处于用户态CPU100%的状态.

DPDK支持的CPU体系架构：x86、ARM、PowerPC（PPC）。

目前来说，DPDK提供的功能还比较初级，要使用DPDK就必须实现ARP、IP层这些基础功能，有一定上手难度。如果要更高层的业务使用或者其他设备使用，还需要用户态的传输协议、设备驱动支持。

## 第三章 数据传输协议NDTP

本章基于现有NTB提供的底层环境，对于NTB传输协议设计过程中面临的各种设计进行了详细的叙述。即在NTB主要应用于机架内Server互连的情况下，多进程如何共享NTB的内存空间、在底层PCIe网络保证了有序且无损的情况下如何设计NTB的消息包头、为了减少内存拷贝加速传输，应使用哪些系统设置，以及对用户进程的发送/接收缓冲区的结构设计以及管理。

### 3.1 NDTP的主要特点

PCIe是一种点对点连接的总线，采用了全双工的传输设计，即允许在同一时刻，同时进行发送和接收数据。一个完整的PCIe体系结构包括应用层、事务层（Transaction Layer）、数据链路层（Data Link Layer）和物理层（Physical Layer）。

NDTP是我们针对NTB传输原理的特点，基于PCIe总线实现的传输协议，对应于PCIe体系中的应用层，以及TCP/IP网络中的TCP层。与TCP协议类似，本文实现的NDTP也是一种可靠的、面向连接的、全双工的协议。

### 3.2 实现NTB的用户态驱动

现有的NTB内核态驱动只提供了简单的设备启动和停止的函数，分配映射的物理内存以及获得分配的物理内存地址功能。其功能较为简单，无法满足NDTP协议对设备控制的需求。出于性能优化，以及开发难度的考虑，我们选择基于DPDK提供的用户态驱动框架，重新实现NTB设备的驱动程序，提供更为完整的控制函数接口。

DPDK已经为其他人开发用户态驱动程序给了一套接口，并且完成了大量底层的工作，基于DPDK的NTB驱动既可以使用DPDK环境下一些比较成熟的库，同时按照其标准暴露出统一的接口也便于后期进行设备管理。

用户态驱动程序UIO（Userspace I/O）是运行在用户空间的I/O技术。Linux系统中的驱动设备普遍都是运行在内核空间的，而在用户空间的应用程序进行调用即可。

 虽然UIO是用户态驱动程序，但在内核部分依然需要进行的一部分工作如下：

（1）分配和记录设备需要的资源和注册UIO设备

（2）必须在内核空间实现的小部分中断应答函数

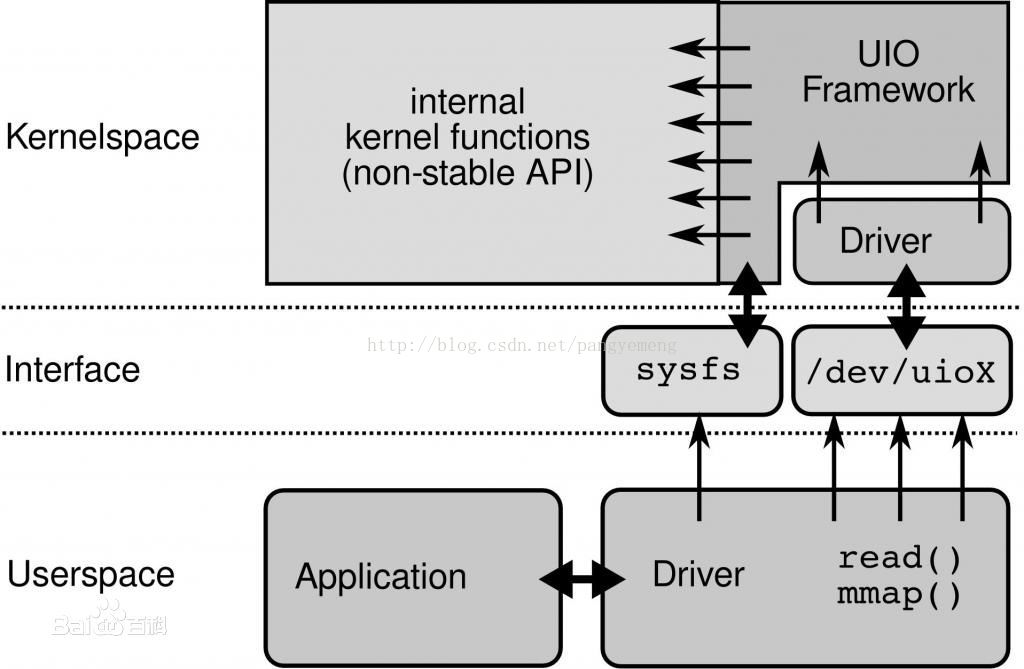


图3.1 DPDK用户态驱动总体架构

如上图3.1所示，基于DPDK开发用户态驱动，对于内核空间的工作来说，DPDK已经提供了一个UIO Framework封装了内核部分的工作，只提供了一些接口供开发用户态驱动的开发者调用，主要包括sysfs用于将系统中的设备组织成层次结构以及在/dev目录下创建一个uio设备,例如/dev/uiox，应用层可以通过read系统调用来访问/dev/uiox设备。

基于DPDK的NTB驱动的优势：

1. 用户空间驱动程序的优点
2. 可以和整个C库链接。
3. 在驱动中可以使用浮点数，在某些特殊的硬件中，可能需要使用浮点数，而Linux内核并不提供浮点数的支持。如果能在用户态实现驱动，就可以轻松解决这一问题。
4. 驱动问题不会导致整个系统挂起。内核态驱动的一些错误常常导致整个系统挂起。
5. 用户态的驱动调试方便。将以前内核模块驱动所需要做的工作移到了用户态空间操作。
6. 方便地使用DPDK提供的底层抽象环境，方便地使用大页内存（hugepage），提高Cache命中率并进行内存共享。

目前DPDK RAW Device 定义的统一管理接口包括以下函数：

struct rte\_rawdev\_ops {

    /\*\*< Get device info. \*/

    rawdev\_info\_get\_t dev\_info\_get;

    /\*\*< Configure device. \*/

    rawdev\_configure\_t dev\_configure;

    /\*\*< Start device. \*/

    rawdev\_start\_t dev\_start;

    /\*\*< Stop device. \*/

    rawdev\_stop\_t dev\_stop;

    /\*\*< Close device. \*/

    rawdev\_close\_t dev\_close;

    /\*\*< Reset device. \*/

    rawdev\_reset\_t dev\_reset;

    /\*\*< Get raw queue configuration. \*/

    rawdev\_queue\_conf\_get\_t queue\_def\_conf;

    /\*\*< Set up an raw queue. \*/

    rawdev\_queue\_setup\_t queue\_setup;

    /\*\*< Release an raw queue. \*/

    rawdev\_queue\_release\_t queue\_release;

    /\*\*< Get the number of queues attached to the device \*/

    rawdev\_queue\_count\_t queue\_count;

    /\*\*< Enqueue an array of raw buffers to device. \*/

    rawdev\_enqueue\_bufs\_t enqueue\_bufs;

    /\*\*< Dequeue an array of raw buffers from device. \*/

    /\*\* TODO: Callback based enqueue and dequeue support \*/

    rawdev\_dequeue\_bufs\_t dequeue\_bufs;

    /\* Dump internal information \*/

    rawdev\_dump\_t dump;

    /\*\*< Get an attribute managed by the implementation \*/

    rawdev\_get\_attr\_t attr\_get;

    /\*\*< Set an attribute managed by the implementation \*/

    rawdev\_set\_attr\_t attr\_set;

    /\*\*< Get extended device statistics. \*/

    rawdev\_xstats\_get\_t xstats\_get;

    /\*\*< Get names of extended stats. \*/

    rawdev\_xstats\_get\_names\_t xstats\_get\_names;

    /\*\*< Get one value by name. \*/

    rawdev\_xstats\_get\_by\_name\_t xstats\_get\_by\_name;

    /\*\*< Reset the statistics values in xstats. \*/

    rawdev\_xstats\_reset\_t xstats\_reset;

    /\*\*< Obtain firmware status \*/

    rawdev\_firmware\_status\_get\_t firmware\_status\_get;

    /\*\*< Obtain firmware version information \*/

    rawdev\_firmware\_version\_get\_t firmware\_version\_get;

    /\*\*< Load firmware \*/

    rawdev\_firmware\_load\_t firmware\_load;

    /\*\*< Unload firmware \*/

    rawdev\_firmware\_unload\_t firmware\_unload;

    /\*\*< Device selftest function \*/

    rawdev\_selftest\_t dev\_selftest;

};

主要功能包括获取设备信息、开启/停止/关闭设备、在PCIe设备内存内创建FIFO队列、获取设备寄存器信息、获取设备状态等。实现了以上接口函数之后，便可使用DPDK drivers提供的统一接口管理NTB设备。

除了以上标准接口所要求的函数之外，我们还实现了一些NTB设备独有的操作函数，根据NTB的传输过程，主要包括更改其映射的物理内存地址、获得NTB设备映射的本地物理内存地址、物理地址空间等信息，Send/Recv相关函数等等。

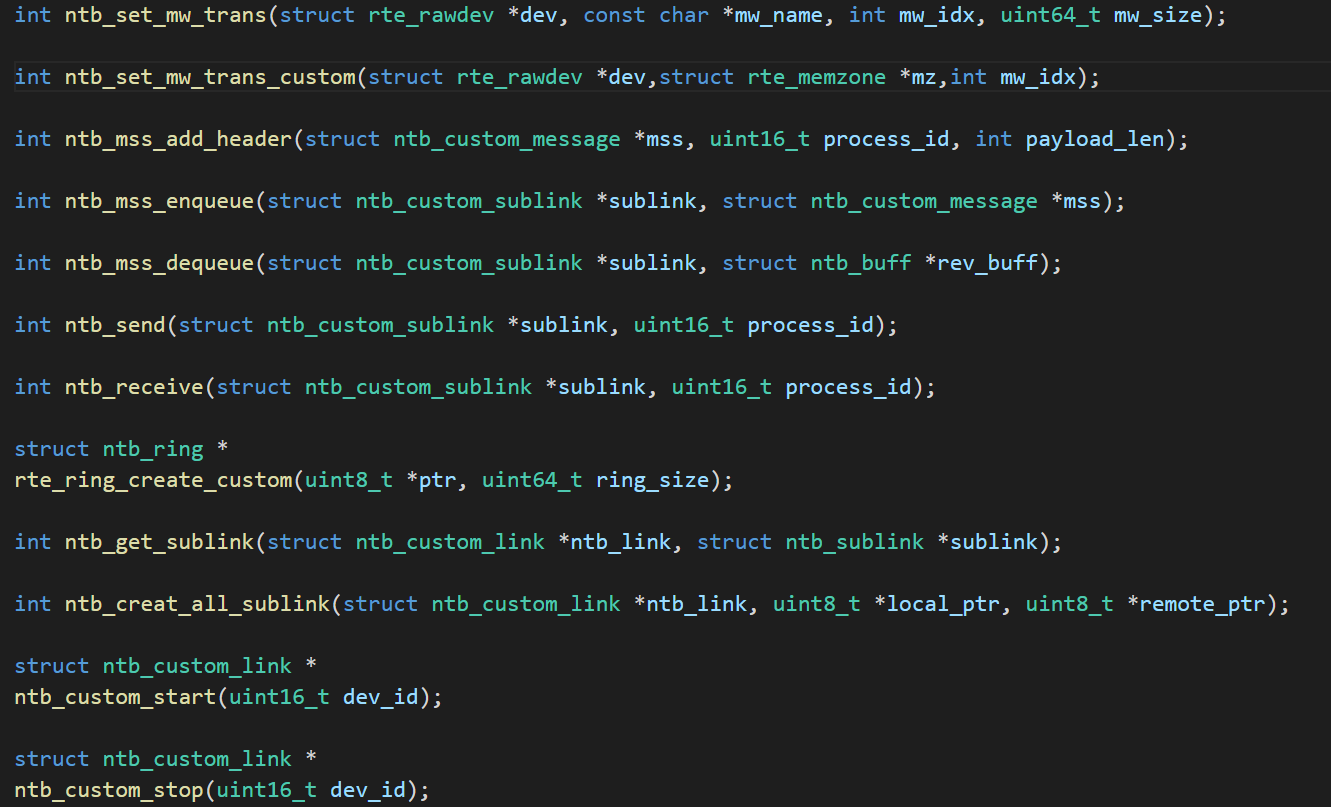


图3.2 NTB独有的操作函数(部分)

### 3.3 NDTP管理Server连接

#### 3.3.1 多物理机管理

DPDK进程使用设备是独占式的，因此基于DPDK设备驱动无法令多个进程同时使用NTB设备，为了让多个进程同时使用NDTP进行通信，需要一个NTB MGT管理模块对设备进行管理，NDTP协议通过实现这个模块对传输过程进行管理。

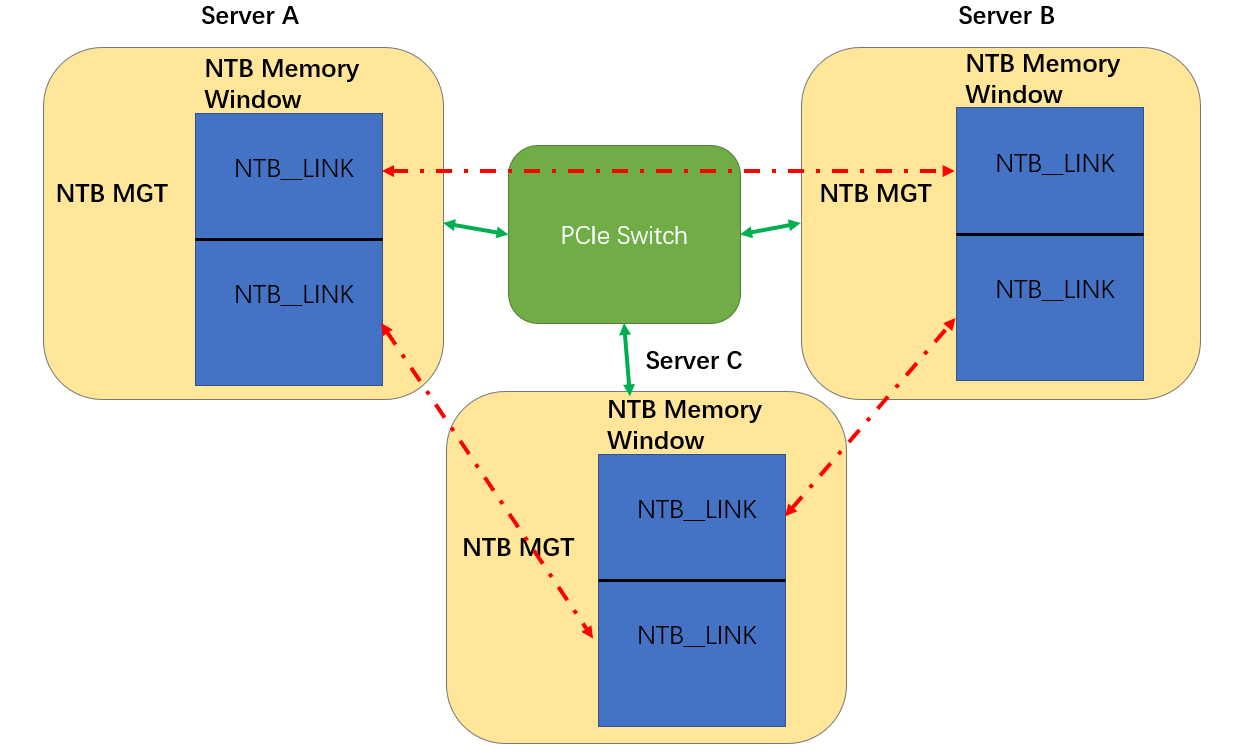


图3.3 NDTP多物理机连接

**NTB MGT：**NTB设备的管理模块

**NTB Link：**位于为NTB的物理地址空间内，代表Server之间的连接，不同的Server之间通过不同的NTB Link相连。所有NTB Link均位于为NTB设备分配的物理内存之中。

当机架内部有n台Server，并且每台Server都将一个NTB设备连接到PCIe Switch上的时候，NTB MGT会将NTB的物理地址空间分成均等的n-1块，每个NTB Link独占其中的一块内存。这样的设计使得多个NTB设备互连时，每个Server之间的连接都有自己独立的内存区域，这样在进行远程DMA的时候可以避免在多个Server之间进行复杂的地址协商过程。

这个设计的缺点是当PCIe Switch中的NTB设备数量过多时，每个NTB\_Link所分得的物理内存会变小，影响传输效率。然而，由于PCI-SIG标准的限制，任何一个系统内PCIe设备不会超过32个，同时由于NDTP的应用场景主要在机架内互连，一个机架内的Server数量不会过多，因此在在实际应用中并不会出现NTB Link所分得的内存过小的问题。

#### 3.3.2 多用户连接支持

因为每个NTB Link代表的是不同Server之间的连接，Server若想创建多个连接并且共享某个特定NTB Link所拥有的内存区域进行数据传输通常有两种方式：

1. 每个连接独占一块内存，创建属于自己的数据缓冲区：

如果对于每个进程创建的连接都要建立一个缓冲区，那么为了防止建立、删除连接的过程中产生内存碎片，那么每个缓冲区所占用的内存大小最好固定。同时为了同时支持尽可能多的进程同时建立连接，每个连接所分配的缓冲区应该尽可能小。但为了不影响传输性能，每个连接分配的缓冲区应该适当增大。由于系统环境可能不同，比较难确定一个合适的缓冲区大小。

同时，由于NTB设备所拥有物理地址空间大小是固定的（目前最大为512MB），若每个连接固定分配8MB缓冲区，那么同时只能够支持512/8 = 64个连接。同时由于本文的NDTP协议其应用场景主要是机架内互连，在一个机架内存在十几台服务器的时候，每台服务器之间平均只支持3到4个进程同时连接。这样会极大的限制这个协议的可用性。因此我们选择使用第二种方式。

1. 每个NTB Link中建立一个数据缓冲队列，多个连接共享这个队列进行数据的传输。

由于多个进程可以共享一个队列，这种设计的优点是可以解除物理内存对连接数量的限制，同时可以支持上万个连接。缺点是比较难进行进程级别的流控。同时由于共享环形队列，需要添加包头来标识该数据包是属于哪个进程的。

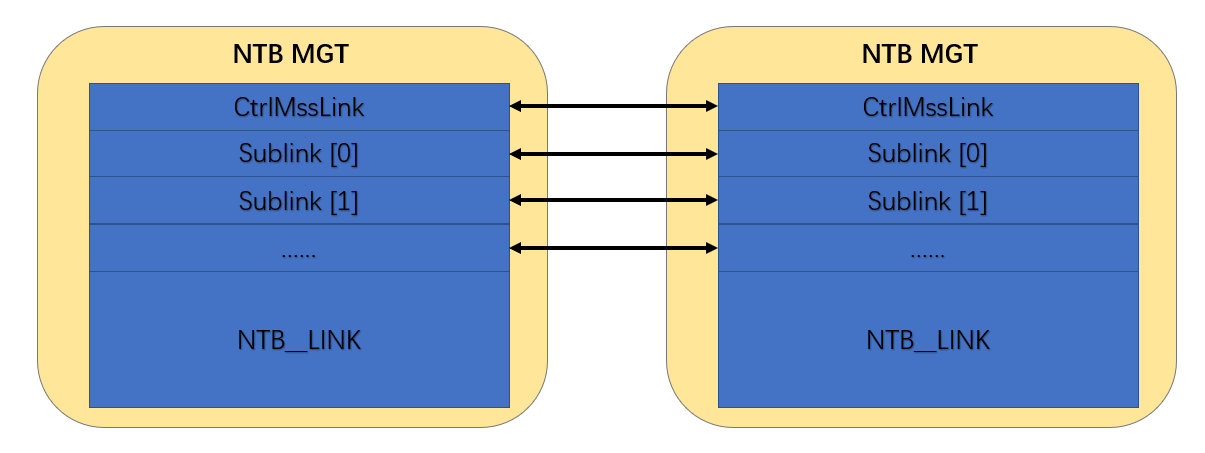


图3.4 NTB\_LINK数据结构

**NTB Sublink：**每个NTB Link里包含8（自定义数量，默认为8）个Sublink。不同的Sublink代表不同的传输优先级，属于同一个优先级的连接使用同一个Sublink，每个Sublink均同时支持多个连接。每个Sublink包含两个NTB RING的指针（NTB RING是一个环形队列），一个指向本地的MSS RING，另一个指向对端的MSS RING。本地向对端的MSS RING中写入NTB Message，读取本地MSS RING的Message。

**CtrlMssLink：**每个NTB Link里包含一个负责传递控制消息的CtrlMssLink，整个NTB Link的控制消息都通过CtrlMssLink中的环形队列传输，其数据结构及原理与NTB Sublink基本相同，区别是环形队列中的Ctrl Message元素与NTB Message相比更小。控制消息包括建立连接、释放连接、Socket发送窗口更新消息等等。

### 3.4 NDTP Socket

NDTP协议作为一个传输层协议，把连接作为最基本的抽象，每一条NDTP的连接由两个端点。这个端点不是主机的IP地址或应用进程，而是Link号和Port号。其功能近似于套接字Socket，我们将其命名为NDTP Socket。NDTP Socket定义为Port号拼接到Link号之后：

NDTP Socket = (NTB\_Link:NTB\_Sublink:Port)

由于NTB的通讯原理，每一个NTB Link都代表着Server之间点对点的连接，不同于TCP、UDP一类的协议，由于不存在多对多的连接，因此无需使用源Port、目的Port分别表示本地和对端的端口号。只使用一个Port号便可以表明使用的是哪一个NTB Socket。

NDTP管理进程会对每个NDTP Socket的状态进行管理，主要包括Occupied标志(Occupied)、发送窗口大小(Send Window)、接收队列指针(Ring \*)、优先级信息等。

#### 3.4.1 NDTP Socket的发送/接收缓冲区

参考TCP协议实现，TCP套接口有一个发送缓冲区，可以用SO\_SNDBUF套接口选项来改变这一缓冲区的大小。当应用进程调用Write往套接口写数据时，内核从应用进程缓冲区中拷贝所有数据到套接口的发送缓冲区，如果套接口发送缓冲区容不下应用程序的所有数据，或者是应用进程的缓冲区大于套接口的发送缓冲区，或者是套接口的发送缓冲区中有别的数据，应用进程将被挂起**。**

类似TCP套接口的缓冲区，为了实现NDTP Socket的发送/接收缓冲区，同时避免多次用户态、内核态拷贝，我们选择使用共享内存的方式进行发送/接收缓冲区的传递。

在 Linux 操作系统上运行内存需求量较大的应用程序时，由于其采用的默认内存页面大小为 4KB，因而将会产生较多 TLB Miss 和缺页中断，从而大大影响应用程序的性能。当操作系统以 2MB 甚至更大作为分页的单位时，将会大大减少 TLB Miss 和缺页中断的数量，显著提高应用程序的性能。这也是提出大页内存(hugepage)的起因。

大页预留之后，接下来则涉及使用的问题。DPDK使用hugetlbfs来使用大页。首先，它需要把大页mount到某个路径 比如 /mnt/huge。接下来，DPDK运行的时候，会使用mmap()系统调用把大页映射到用户态的虚拟地址空间，然后就可以正常使用了。

实现NTB预留的地址空间与实际物理内存的映射也需要用到大页内存，NTB需要使用一块连续的大页内存块进行映射，例如文中NTB在地址空间中预留的内存长度为512MB，因此需要在DPDK预留的大页内存中申请512MB并将起始地址填入NTB的配置寄存器中。

我们设计了Shared Memory MGT模块，申请一块共享的大页(hugepage)内存，并建立一些数据结构管理这块共享内存。应用程序所申请的发送/接收缓冲区均位于这块共享内存中。

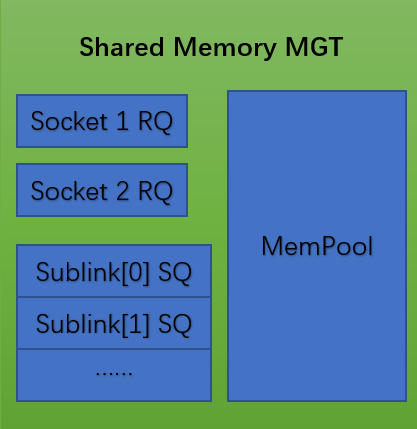


图3.5 Shared Memory MGT结构

上图中涉及了一些对这块共享内存进行管理的数据结构，主要包括一种支持多生产者、单消费者的无锁环形缓冲队列(上图中所有队列均为这种结构)，以及一个内存池。NDTP Socket只有单独的接收缓冲队列RQ，没有单独的发送缓冲队列SQ，而是由每个Sublink中的全部NDTP Socket连接共享这个Sublink的发送队列，所有待发送数据都需要进入这个Sublink Send Queue中，队列中的元素内容为端口号以及一个指向MemPool中内存节点的指针。

#### 3.4.2 Socket缓冲区的无锁环形队列

该数据结构允许管理应用程序的接收队列以及每个Sublink所属的发送队列，队列的元素为指向MemPool中MemNode的指针。Ring不是无限大小的链表，而且具有以下属性：(1)FIFO (2)最大大小是固定的，指针存储在表中 (3)无锁实施 (4)单一消费者出队、多生产者入队 (5)支持批量入队/批量出队

此数据结构相对于链表队列的优点如下：

（1）快速，只需要一个CAS的比较和交换指令，而不是几个双比较和交换指令。（2）比完全无锁队列的实现更加简单。（3）适用于批量入队/出队运营。由于指针存储在表中，因此几个对象的出列不会产生与链接队列中一样多的高速缓存未命中。此外，许多对象的批量出列并不比简单对象的出列成本高。

缺点如下：

（1）尺寸是固定的。（2）在内存方面的成本高于链表队列，空环至少包含N个对象。

下图是Ring的简化表示显示在消费者和生产者头部和尾部指针中，指向存储在数据结构中的对象。

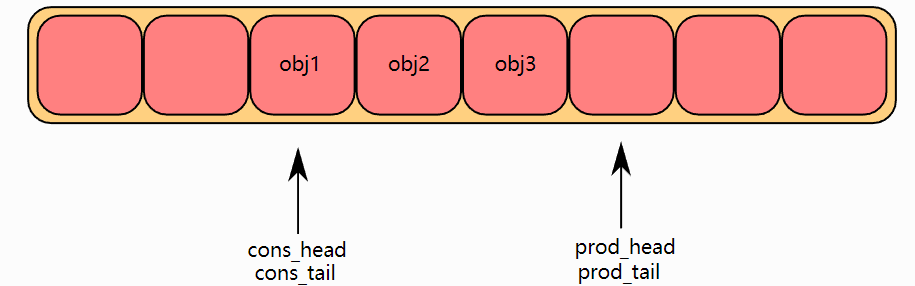


图3.6 Ring结构

环结构由两个头尾耦合组成; 一个由生产者使用，一个由消费者使用。以下部分的数字将它们称为prod\_head，prod\_tail，cons\_head和cons\_tail。

图3.9表示环的简化状态，其是循环缓冲区。函数局部变量的内容表示在图的顶部，环结构的内容表示在图的底部。

（1）单一消费者出队

当消费者从环中取出对象时发生的情况。在此示例中，仅修改了消费者头部和尾部（cons\_head和cons\_tail），并且只有一个使用者。

初始状态是指向同一位置的cons\_head和cons\_tail。

首先，将ring-> cons\_head和ring-> prod\_tail复制到局部变量中。cons\_next局部变量指向表的下一个元素，或者在批量出列的情况下指向几个元素。如果环中没有足够的对象（通过检查prod\_tail检测到这种情况），则会返回错误。

第二步是修改环结构中的ring-> cons\_head以指向与cons\_next相同的位置。指向出列对象（obj1）的指针被复制到用户给出的指针中。

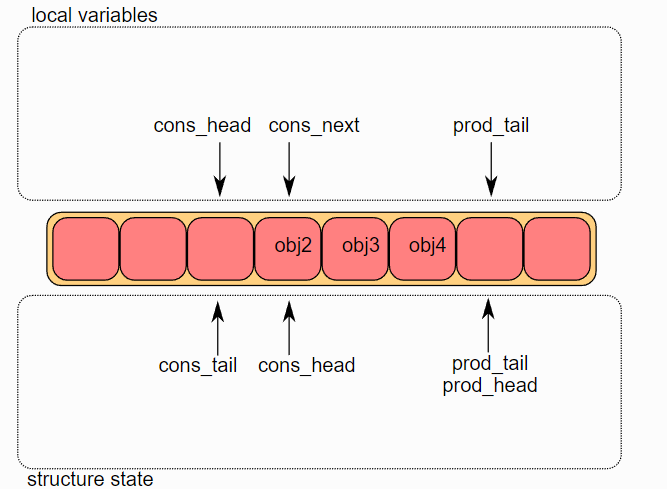


图3.7 单一消费者出队

最后，修改环结构中的ring-> cons\_tail以指向与ring-> cons\_head相同的位置。出列操作完成。

（3）多个生产者入队

本节介绍两个生产者同时向环添加对象时发生的情况。在此示例中，仅修改了生产者头部和尾部（prod\_head和prod\_tail）。

最初的状态是将prod\_head和prod\_tail指向同一位置。

在两个核心上，ring-> prod\_head和ring-> cons\_tail被复制到局部变量中。prod\_next局部变量指向表的下一个元素，或者在批量入队的情况下指向几个元素。

如果环中没有足够的空间（通过检查cons\_tail检测到），则返回错误。

第二步是修改环结构中的ring-> prod\_head，使其指向与prod\_next相同的位置。此操作使用比较和交换（CAS）指令完成，该指令以原子方式执行以下操作：

如果ring-> prod\_head与局部变量prod\_head不同，则CAS操作失败，并且代码在第一步重新启动。

否则，ring-> prod\_head设置为本地prod\_next，CAS操作成功，处理继续。

在图中，操作在核心1上成功，第一步在核心2上重新启动。

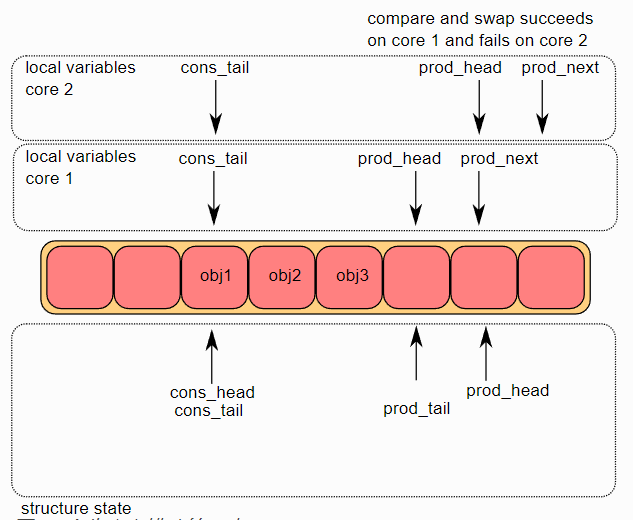


图3.8 多个消费者入队 *(1)*

CAS操作在核心2上重试成功。

核1更新环的一个元素（obj4），核2更新另一个元素（obj5）。

现在每个核心都想更新ring-> prod\_tail。如果ring-> prod\_tail等于prod\_head局部变量，则核心只能更新它。这仅适用于核心1.操作在核心1上完成。

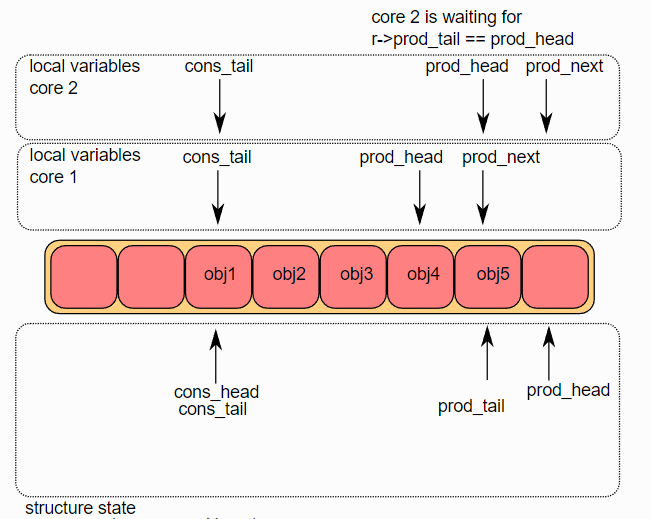


图3.9 多个消费者入队 *(2)*

一旦ring-> prod\_tail由核心1更新，核心2也允许更新它。核心2上的操作也已完成。

#### 3.4.3管理缓冲区内存节点的MemPool

经典的内存池（MemPool）技术，是一种用于分配大量大小相同的小对象的技术。通过该技术可以极大加快内存分配/释放过程。

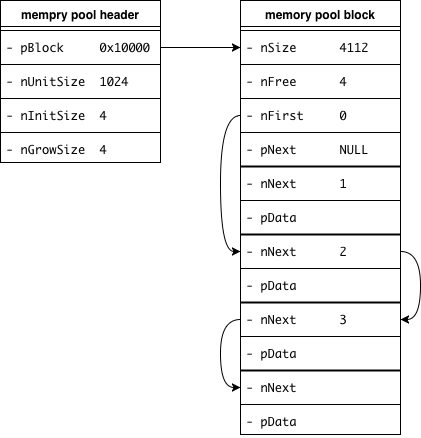


图3.10 一个典型的MemPool结构

内存池在申请了一些内存节点(MemNode)之后，在结构体中保存两个链表，其中一个链表记录自由的内存节点，另一个链表记录已使用的内存结点。以便通过它们可以使用、释放所有申请的内存。

内存申请过程分为两种情况：

在自由内存结点链表（FreeNodeList）非空。在此情况下，Alloc过程只是从链表中摘下一个结点的过程。

否则，意味着NDTP MGT代表的管理进程申请的共享内存耗尽，进行内存申请的应用程序需要挂起。

在NDTP MGT管理进程与应用程序的共享内存中创建内存池，将该块共享内存划分为多个内存节点，每个内存节点头部存在一个表明该节点有效数据长度的Header。我们选择使用Ring结构管理应用程序的发送/接收缓冲区，除了在共享内存中创建内存池(MemPool)结构外，每个应用程序还会创建一个发送队列和一个接收队列（为Ring结构），队列中的对象是指向内存池中内存节点的一个指针。

### 3.5 NDTP的流量控制

由于NDTP协议底层的PCIe总线提供了一个无损且有序的可靠网络环境，数据包按序到达且不存在丢包情况。而本文暂且不考虑PCIe物理线路连接断开导致的数据包丢失情况。

因此为了保证可靠性，NDTP协议目前需要的做的是连接的流量控制，保证NTB Sublink中的环形缓冲区以及Shared Memory中各个端口的缓冲队列不会溢出。

#### 3.5.1 NTB Sublink流量控制

NTB Sublink间的传输是通过两个环形队列实现的，NTB MGT模块中保存有每个NTB Link和Sublink的信息，这些信息在NDTP MGT管理程序启动的时候进行初始化。NTB Sublink的基本数据结构如下图所示：

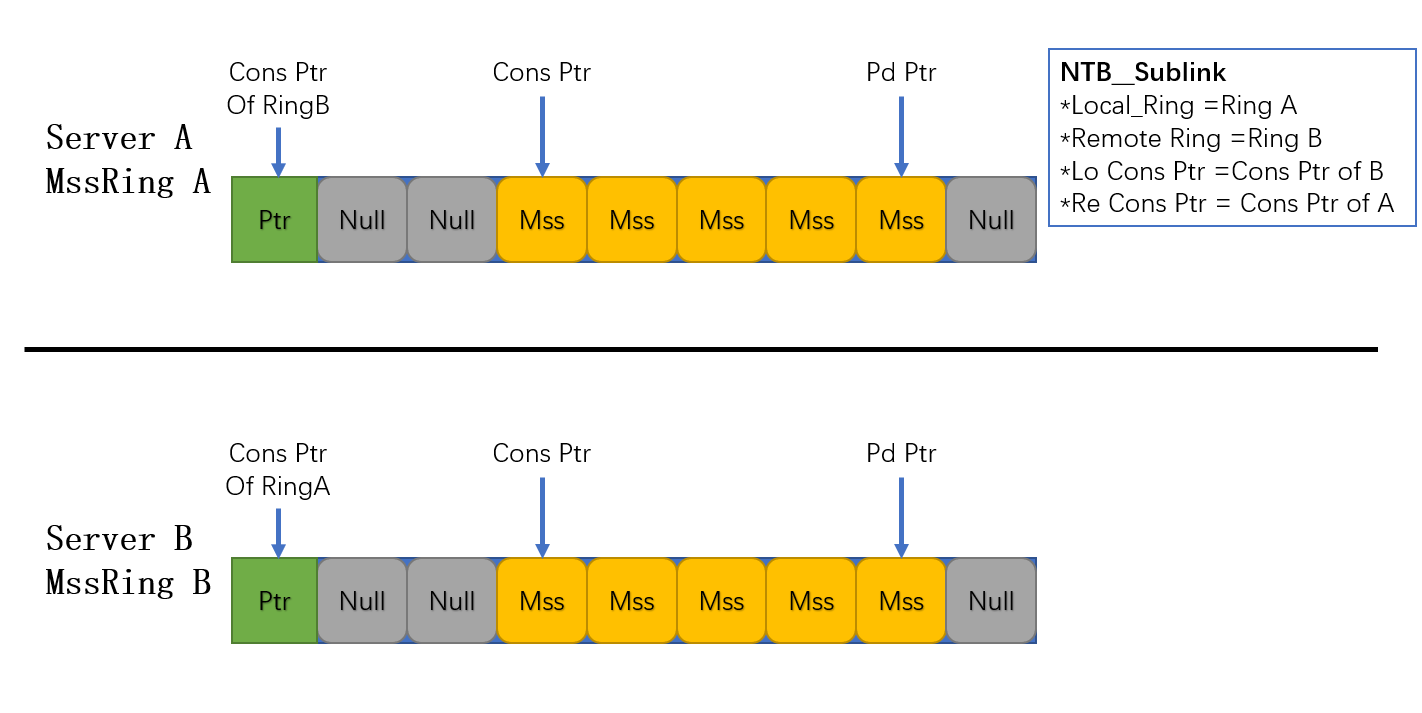


图3.11 MSS\_RING以及Sublink数据结构

Message Ring是基于数组实现的，每一个元素均为NTB Message，其数组大小在初始化时指定，不能动态改变。每个Message Ring均需要有一个生产者指针Cons Ptr以及一个消费者指针Pd Ptr。同时在每个Message Ring的头部增加了一个Ptr元素，存储的是一个64bit的地址，其值等于对端的Cons Ptr指针。

以Server A为例，每个NTB Sublink中持有两个Message Ring的内存地址Local Ring和Remote Ring，其中Local Ring是位于Server A分配给本地NTB设备的物理内存中的MssRing A，负责接收Server B发送过来的NTB Message，对于Local Ring，Server A只扮演一个消费者的角色，Server A的NTB Sublink结构体中只保存Local Ring的Cons Ptr地址。

Remote Ring通过NTB映射到位于对端Server B物理内存中的MssRing B，对于对端的Remote Ring，本地只扮演一个生产者，本地向Remote Ring中生产者指针(Pd Ptr)所指的位置写数据，通过NTB的地址翻译，即可到达Server B物理内存的MssRing中。

为了防止环形队列的溢出，Server A在向Server B中MssRing的生产者指针地址写入数据并对指针地址进行累加操作前，需要知道Server B中消费者指针Cons Ptr所指的位置，防止写入越界，覆盖掉Server B尚未读取的信息。由于NTB设备的远程写入(Remote Write)性能远高于远程读取(Remote Read)性能，因此我们希望Server B可以将自身的消费者指针Cons Ptr写入到Server A的物理内存中，避免Server A进行远程读取操作。为了避免NTB通信中小消息过多的情况，Server B没读取一定数量的NTB Message之后才会进行一次远程写操作，来更新Server A中Ptr元素的值。

#### 3.5.2 NDTP Socket的流量控制

由于NDTP Socket的接收缓冲队列位于NDTP MGT进程所申请的共享内存中，队列中的元素为指向MemPool中一个内存节点的指针。这个MemPool是由所有NDTP Socket连接共享的。如果不对Socket连接的发送速率加以控制，一个Socket占用了过多的内存节点，会导致内存节点不够，从而阻塞其他的发送进程。

类似于TCP协议，利用发送窗口可以方便地在Socket连接层面实现对发送方的流量控制。不同的是NDTP Socket的发送窗口大小不是以字节为单位，而是以MemPool中的内存节点为单位的。并且由于底层网络是可靠的，发送出去的数据并不会返回确认信息，因此发送窗口的含义仅仅是目前还可以发送多少个内存节点的数据，并不像TCP协议一样在发送后未收到确认的部分会保留副本以便进行重传。

一个使用Sublink[0]的NDTP Socket 1发送数据的过程如下：

1. 向MemPool申请一块内存节点，填入数据调用Sublink[0] SQ队列提供的接口，将这个内存节点以及自身的Port号放入待发送队列中，同时NDTP Socket的发送窗口值-=1，当窗口值为0时，函数调用失败，无法进入待发送队列。
2. NDTP MGT进程中的发送线程将Sublink SQ中的数据封装成NDTP Mss发送到另一端的Mss Ring中，另一端的读取线程拆包并且读取数据，并将这些数据写入到向MemPool申请的内存节点中，并将这个内存节点的指针放入Socket 1 RQ中。
3. 对端的Socket 1读取RQ中的数据，在读取了一定数量的内存节点后，调用控制消息的发送函数，将更新发送窗口的消息发送到本地的CtrlMssLink中。
4. 本地收到CtrlMssLink中的消息，更新对应Socket连接的发送窗口值。

### 3.6 NDTP Message结构设计

NDTP Message的包头设计如下图3.12所示：

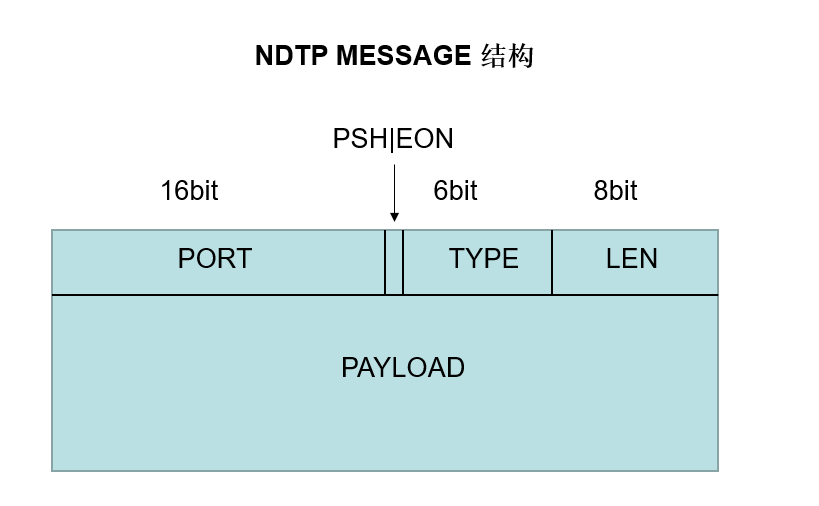


图3.12 MSS\_RING以及NDTP数据包结构

**PORT：**每个LINK同时支持2^16个连接，这些连接的占用状态，以及连接的进程的接收缓冲区、传输优先级、发送窗口大小均记录在NDTP MGT的管理进程中中。由于是点对点连接，所以不像TCP端口一样源端口、目的端口不同。一个端口号，以及其位于的Link号和Sublink号即可唯一的确定其NDTP Socket。

**TYPE**：代表不同的消息类型，目前只有为1.数据类型，以及部分连接控制类型，尚未有更细致的划分。

**PSH标志位**：当接收端收到PSH为1的数据包时，立即将本地Message Ring的消费者指针传递到对端。

**EON标志位**：End of MemNode，表明这是对端发送的一块MemNode中的最后一个数据包。

**LEN**：消息长度，包含头部长度，因此最小为4，接收线程通过LEN的值是否为0来判断Message Ring此处是否

**PAYLOAD：**每个NDTP Message最大包长为128B，PAYLOAD长度为LEN减去头部长度（固定为4B）。

相较于TCP协议，由于NTB基于的PCIe总线协议已经保证了按序到达且不丢包，因此对于NDTP，很多保证可靠传输的字段可以舍弃。

#### 3.6.1 NDTP Message MTU

PCIe总线协议其事务层（Transaction Layer）的TLP包存在Max Payload Size限制，NTB设备硬件上限制的Max Payload Size默认为128B，而目前 Intel AVX512指令集，调用拷贝函数的单次拷贝长度为64B，这样会使得每个TLP包Payload为64B，无法到达Max Payload Size，造成性能损失。为了使NTB设备的性能得到充分利用。

我们使用写合并（Write-Combining）技术，使得每次发送一个Payload Size为128B的PCIe TLP包。写合并是一种计算机总线技术，允许将数据组合并临时存储在缓冲区——写入组合缓冲区（WCB）中 ，并且稍后以突发模式一起释放，而不是单独写入（立即）比特或小块。由于弱排序，写合并不能用于一般存储器访问（数据或代码区域）。写合并不保证写入和读取的组合按预期顺序完成。例如，对特定地址的写/读/写组合将导致读/写/写的写合并顺序，这可能导致在第一次读取时获得错误的值（这可能依赖于之前的写入）。

为了避免上述的读/写顺序问题，NTB读取数据时完全不依赖于之前的写入，并且不需要强排序（总是正确的）之类的场景中。

开启Write Combining需要已知NTB设备被系统分配的地址空间起始地址和长度，如下例子是NTB设备起始地址为0x387fe8000000，长度为512MB情况下开启Write Combing的命令：

echo "disable=1" >> /proc/mtrr

echo "base=0x387fe8000000 size=0x20000000 type=write-combining" >> /proc/mtrr

开启Write Combining后，NDTP的单个TLP包的 Payload Size由64B提高到了128B，在PPS(Packets Per Second)不变的情况下，Remote Write性能从5Gb/s提升到了10Gb/s，详细测试数据见5.3.1节。

若NTB Message最大包长超过128B，则在TLP层需要分成多个PCIE包到达对方内存，存在包头已经写入，但内容尚未写完，对方就已经读取的情况。这种情况虽然可以通过对分片后的包每128B设置一个标志位或者在包尾增加EOF标志来减少一些封包开销，但这样依然是一种以128B为最大包长的行为，并且有可能造成对某一块内存的循环读取。

由上述原因，选择128B为NDTP Message MTU是一个较好的方案。

## 第四章 NDTP管理进程架构设计

本章将前文各部分提到的各部分的技术选择以及协议设计整合起来，设计了一个适用于机架内服务器互连状况下的使用NTB通讯的传输协议NDTP的总体架构。下面对于该协议的总体架构以及协议的工作流程做进一步的阐述。

### 4.1协议总体架构

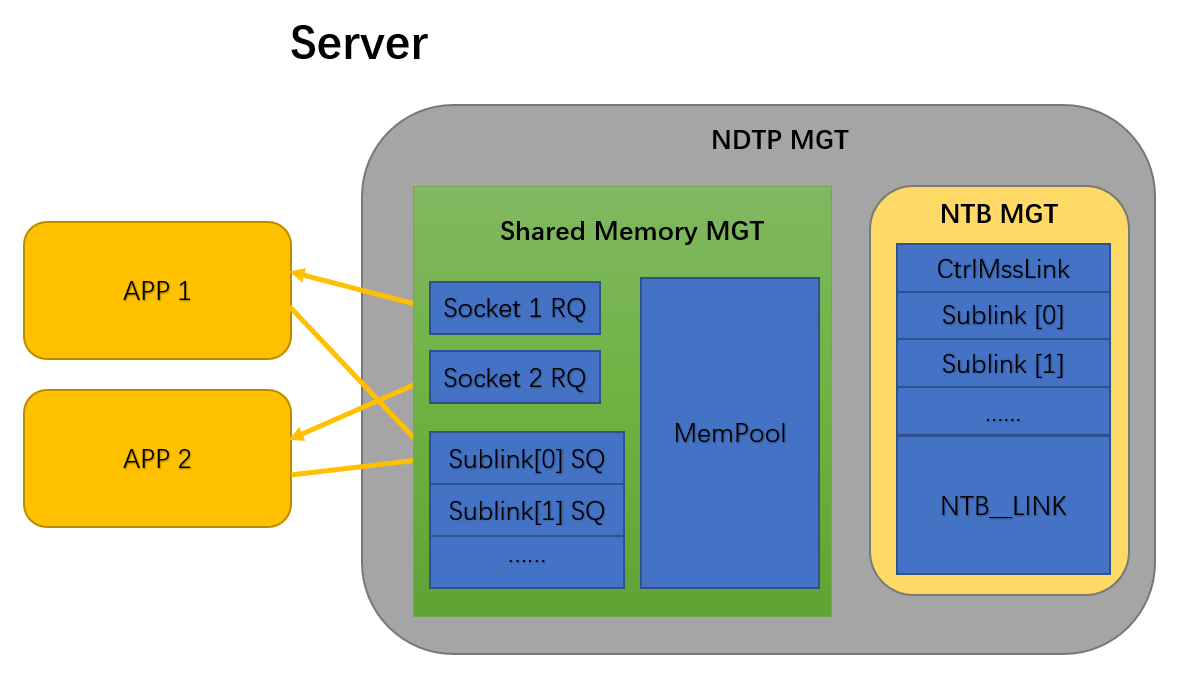


图4.1 NDTP总体架构

本系统主要由发送/接受数据的应用程序APP、NTB传输协议NDTP的管理进程NDTP MGT，以及NDTP MGT内部管理与应用程序的共享内存的Shared Memory MGT管理模块，管理NTB设备以及其与其他Server 的NTB 设备共享的物理地址空间的NTB MGT管理模块。

应用程序APP1/2的发送/接收缓冲队列均位于NDTP MGT进程的共享内存中。

NTB MGT为管理NTB硬件的守护进程，所有传输操作通过它暴露的接口函数完成。

### 4.2 模块设计及功能说明

#### 4.2.1 NDTP管理模块

为了管理Server中的NDTP协议正常稳定的运行，我们需要一个管理进程对协议运行的整体情况进行把握。因此我们设计了NDTP MGT模块来统管整个协议所使用的各种资源，以及监控协议的运行状况，记录了全部NDTP Socket的连接信息。

NDTP MGT中主要包括两个模块，一个是Shared Memory MGT，主要负责管理与应用程序进行数据传递的共享内存；另一个是NTB MGT，基于NTB的用户态驱动对NTB两端的物理地址空间进行管理。

#### 4.2.2 Shared Memory MGT模块

Shared Memory MGT模块主要负责管理NDTP MGT申请的共享内存中大页内存的管理，以及登记相关数据结构，进行NDTP MGT与用户进程间的通讯。

Shared Memory MGT是基于DPDK的环境抽象层(Environment Abstraction Layer, EAL)构建的，环境抽象层提供了系统大页内存的抽象，并且提供了对建立的数据结构进行命名、查找的功能。这样用户进程只要知道自身建立的数据结构的名称，便可以通过Shared Memory MGT获得指向该数据结构的指针。

Shared Memory MGT模块主要由以下部分构成：

1. Socket RQ

每个Socket连接均在Shared Memory中独占式拥有一个环形接收队列RQ，队列的大小在建立连接时由用户进程指定。NTB MGT进程读取Sublink中的数据后，向MemPool中申请一个内存节点并将数据存储到其中，之后将这个内存节点的头指针放入对应Socket的RQ队列，等待用户进程读取。

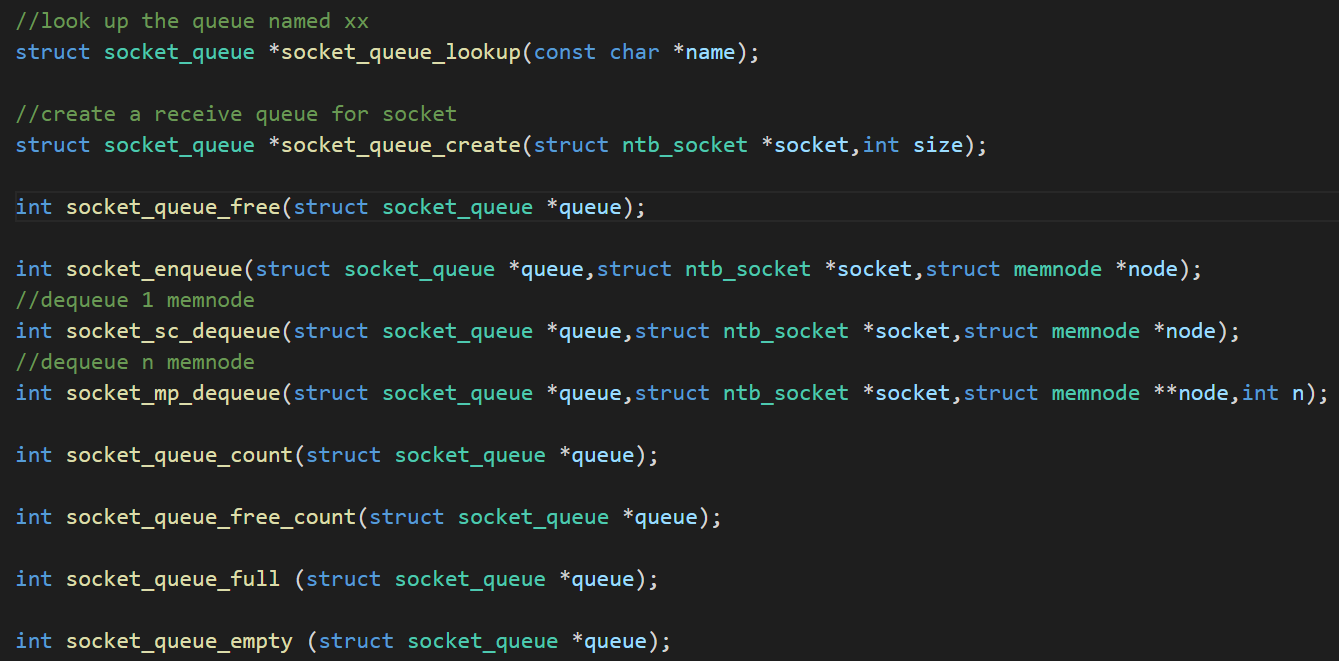


图4.2 环形队列Queue部分函数定义

1. Sublink SQ

每个Sublink共享一个发送缓冲队列SQ，由3.4.2节可知，本队列是一个支持多生产者单消费者的无锁环形队列。每个Sublink中的多个NDTP Socket向MemPool中申请发送缓冲节点并写入数据后，可以并发地向SQ发起入队操作。

1. MemPool

在Shared Memory MGT中创建内存池，将该块共享内存划分为多个内存节点，内存节点的大小和数量在NDTP MGT进程启动时进行相关的配置，每个内存节点头部存在一个表明该节点有效数据长度的Header。

#### 4.2.3 NTB MGT模块

NDTP MGT进程启动后，会启动NTB MGT模块，对NTB设备本地和对端的地址空间进行分配，记录NTB Link/Sublink中的信息，并管理发送和接收线程，并拥有本模块的流量控制机制。

NTB MGT模块在NDTP管理进程启动后优先启动，并遍历PCIe Swtich中所连接的NTB设备，假如除了自身NTB外，目前的PCIe Switch中还存在n个NTB设备，那么NTB MGT模块会将系统分配的NTB Memory Window平均分成n份，按照遍历的顺序分别分配给n个NTB\_Link，每个NTB Link对应一条B2B的通过NTB连接的物理机。

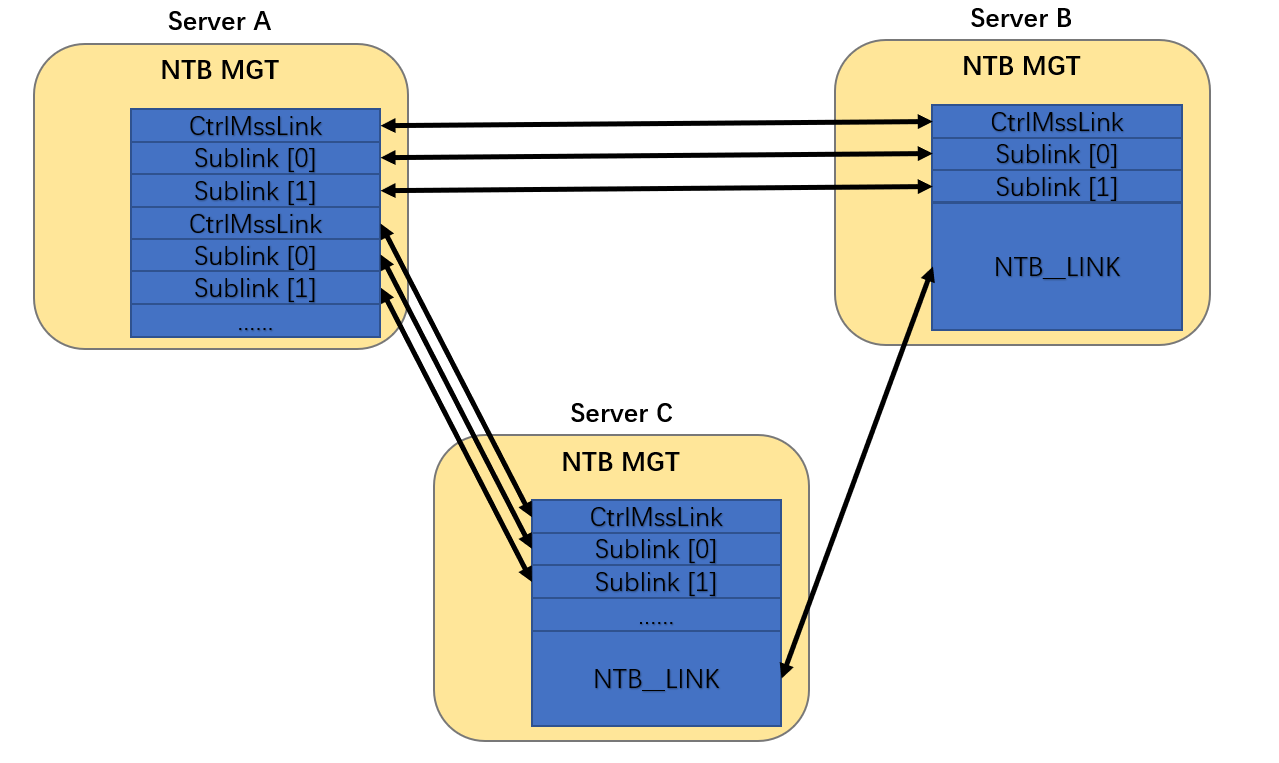


图4.3 物理机间NTB\_LINK、Sublink结构

因为NDTP的数据包是顺序写入环形缓冲区中的，因此如果参照TCP的发送窗口流控的实现，必须CPU顺序读取到的时候才可以更新流控信息，这其中会带来一定的时延，导致控制信息更新不及时，可能会造成意料之外的阻塞。如两端同时进行远程写操作，写窗口消耗干净，并同时发送了ack包更新写窗口，等待按顺序读取到ack包时才会更新写窗口，期间两端的写入操作都会停止，继续写操作，这样就带来了NTB传输的性能损耗。此外将流控信息加在包头还会导致额外的CPU开销，同时每个消息所能携带的Payload长度变短，传输效率降低。

因此单独建立一个Cons Ptr的内存区域就十分必要了，由于NTB MGT的设计，不允许多个线程同时使用Sublink，即同一时刻内只能有一个线程使用PCIe总线传输Cons Ptr，因此一个单独的Cons Ptr区域存储此信息不会产生错误。

### 4.3 协议传输过程

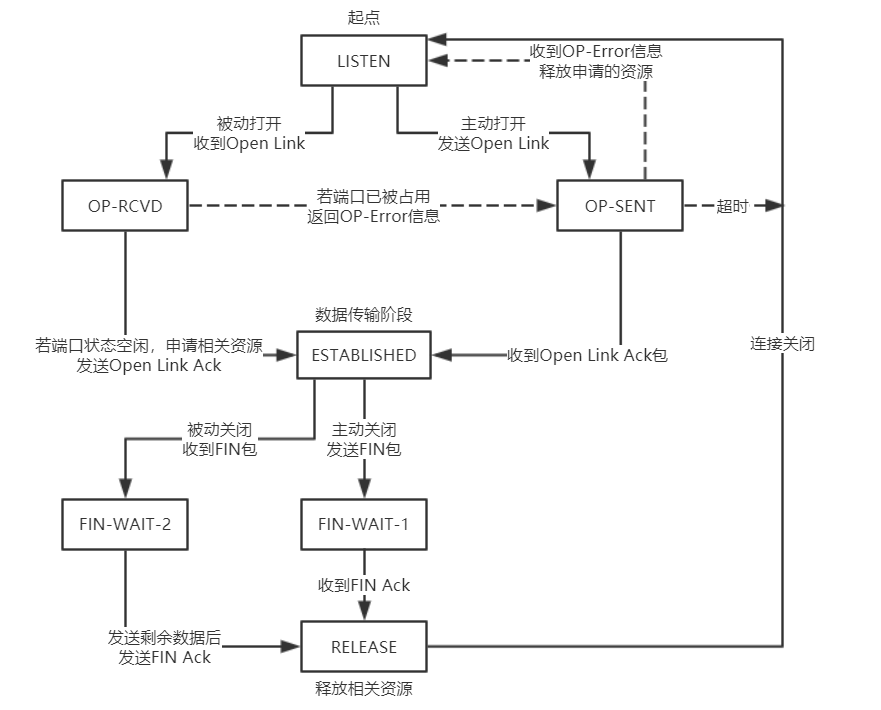


图4.4 NDTP有限状态机

#### 4.3.1 NDTP MGT进程的启动

(1)初始化DPDK的环境抽象层EAL，把系统预留的大页内存挂载到某个路径比如 /mnt/huge。接下来，使用mmap()系统调用把大页映射到用户态的虚拟地址空间，从这部分内存中申请512MB交予NTB MGT模块使用。剩余的大页内存交予Shared Memory MGT模块进行管理。

(2)NTB MGT模块将获得的512MB连续物理内存的起始地址填入NTB设备的翻译寄存器，结束地址填入限制寄存器。并根据PCIe Switch中遍历到的设备数量以及相关配置参数进行各个NTB Link、NTB Sublink的内存区域的分配。

(3) Shared Memory MGT在拥有的大页内存中建立MemPool，同时根据NTB MGT模块创建的NTB Link、NTB Sublink的数量进行Sublink SQ的创建，每个Sublink SQ都拥有自己的名称作为Key值，便于查找，每个Sublink SQ与NTB Sublink一一对应。同时必须预留一部分空间为NDTP Socket建立接收队列，每建立一个NDTP Socket连接，Shared Memory MGT中必须建立一个Socket RQ与其对应。当Shared Memory空间不足时建立失败，无法再建立连接。

#### 4.3.2 NDTP Socket连接的建立

(1) Server A中进程选择使用的NTB LINK号、NTB Sublink号（Sublink的含义为优先级）、使用的Port号，调用相关的连接建立函数DialNDTP并将相关的参数传入。

(2) DialNDTP根据Shared Memory MGT中的信息判断该NDTP Socket是否被占用，若被占用返回error。若没被占用，则发送一个MSS\_TYPE为OPEN-LINK的消息并写入相关的参数发送到Server B端的CtrlMssLink中。

(3)Server B读取到本地CtrlMssLink中的OPEN-LINK消息后，判断该NDTP Socket是否被占用，若占用则向Server A中CtrlMssLink写入error消息。没被占用则将该Socket设置为占用状态，并写入相关的配置信息，之后在Server B的Shared Memory中建立接受队列Socket SQ。并返回MSS\_TYPE为OPEN-LINK-ACK的消息。

(4)Server A读取到本地CtrlMssLink中的OPEN-LINK-ACK消息后，建立接收队列Socket RQ，将其指针写入Sublink中的Port信息中，同时也将这两个指针传递给应用程序，使其可以向其中传递/读取其中所指向的MemPool中的内存块。

#### 4.3.3 NDTP Socket连接的释放

(1) NDTP默认均为长连接，不会主动释放现有的Socket连接，需由应用进程调用相关函数进行释放。调用释放链接函数后，发送队列所指向的内存块的数据发送完毕后，发送一个FIN包。FIN包只能由Client端发送。之后释放发送队列。

(2) 接收线程到FIN包后，根据其Socket号，向其Sublink SQ中入列一个FIN-ACK包，由于SQ的发送是顺序且可靠的，因此当FIN-ACK包到达对端时。该NDTP Socket所欲发送的内容必然已经全部发送完毕，收到FIN-ACK包的一端释放相关缓冲区。之后修改Daemon中该端口占用状态。

#### 4.3.4 应用程序数据发送过程

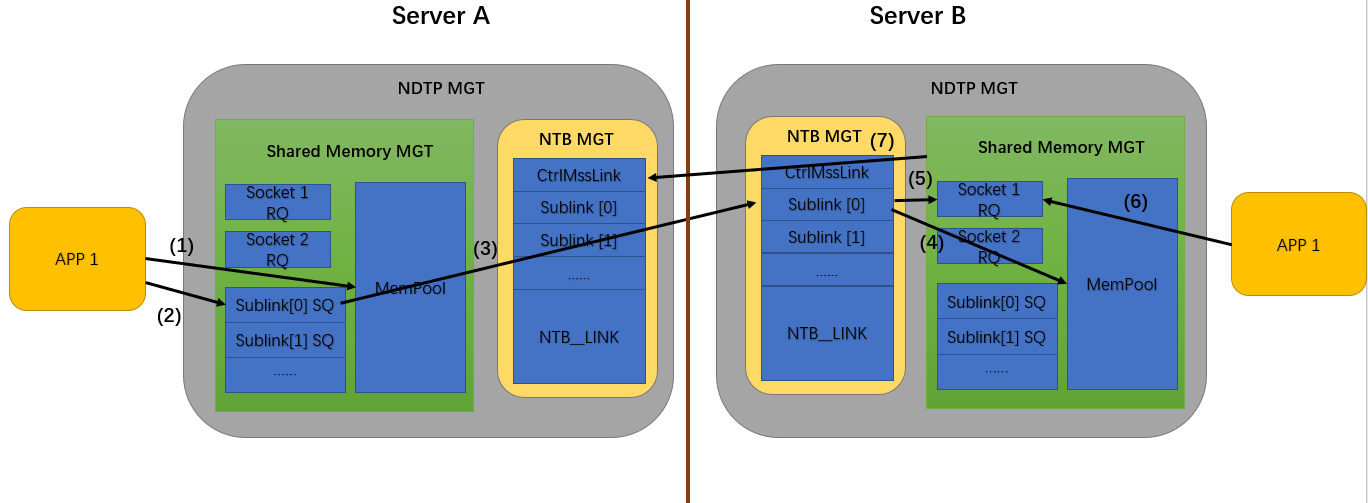


图4.5 应用程序通过NDTP发送数据的过程

目前由于NTB用户态驱动程序、以及NDTP MGT管理进程均为基于DPDK构建，为了便于与NDTP MGT进程进行资源共享和同步，并且更方便的使用DPDK提供的抽象环境。目前的应用需要基于DPDK重新构建自己的发送程序。NDTP MGT已经为应用程序提供了相关的Send/Reve函数和连接管理函数。

(1)建立连接后，通过Socket 1进行连接的应用程序APP 1向共享内存管理模块Shared Memory MGT中的MemPool结构申请一个内存节点(MemNode)，作为自身的发送缓冲区。

(2)写入完毕后将这个发送缓冲区的头部指针写入使用的NDTP Socket对应的发送队列Sublink SQ中，同时Shared Memory MGT中记录的该Socket发送窗口减1。

(3)NTB MGT中的发送线程按照优先级顺序轮询所有Shared Memory MGT模块中登记的Sublink SQ，当遍历到的SQ不为空时，进行其中的内存节点进行出列操作，对其中的数据进行封包并向对端Sublink的MssRing中发送。发送完毕后释放该内存节点。

(4)、(5)对端的接收线程按照优先级轮询NTB Sublink所属的物理内存中的MssRing，若不为空，则进行解包，并根据其NDTP Socket号，向共享内存的MemPool结构申请一个内存节点作为接收缓冲区，向接收缓冲区中写入数据后，将这个接收缓冲区的头部指针写入其Socket号对应的接收队列Socket RQ中。

(6)对端的应用程序读取自身所属的Socket RQ中所指向的内存节点中的数据，之后将该内存节点释放。

(7)在某一个Socket释放了一定数量个内存节点后，Shared Memory MGT会向对端CtrlMssLink中发送一条更新发送窗口的消息。对端收到后会对该Socket的发送窗口进行更新。

## 第五章 实验与分析

### 5.1 实验环境介绍

对于本文中所有测试的实验环境，其设备信息如下：

测试环境：

|  |  |
| --- | --- |
| CPU | Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz |
| Memory | 48GB DDR4 LRDIMM 2666MHz |
| Linux Kernel version | 5.1.3-050103-generic |
| NTB Device Information | PCI-E Gen 3 Width x4 Max Payload Size = 128B |
| Network Adapter | Intel Corporation Ethernet Connection X722 for 10GBASE-T |
| RDMA Adapter | MCX516A-CDAT ConnectX-5 (100Gbps) |

### 5.2 实验环境配置

在服务器的bios中设置 NTB PCIe PORT 选择 NTB to NTB ；Enable NTB bars 选择 Enabled ；BAR size 设置为29；

之后为dpdk程序设置1GB大页，mount -t hugetlbfs -o pagesize=1GB nodev /mnt/huge\_1GB/

echo 1 > /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr\_hugepages

echo 1 > /sys/devices/system/node/node1/hugepages/hugepages-1048576kB/nr\_hugepages

查看free大页数 cat /sys/kernel/mm/hugepages/hugepages-1048576kB/free\_hugepages

mount | grep huge查看大页目录

每次重启服务器之后需要：设置环境变量： export RTE\_SDK=/home/ntb-server1/dpdk

编译dpdk 库： RTE\_SDK/usertools/dpdk-setup.py

插入模块： modprobe iGb\_uio

绑定ntb设备 ： RTE\_SDK/usertools/dpdk-devbind.py --bind=uio\_pci\_generic 17:00.0

设置NTB分配的内存为Write-combining： echo "disable=1" >> /proc/mtrr

echo "base=0x387fff400000 size=0x400000 type=write-combining" >> /proc/mtrr

### 5.3 系统性能测试

#### 5.3.1 NTB顺序读写性能

1. 测试内容
   1. 针对单核下开启/关闭GNU -O3编译优化进行测试
   2. 针对单核下Memcpy函数R/W内存长度不同的情况进行测试
   3. 针对多核情况下内存使用Write-Back/Write-Combining模式的R/W性能进行测试
2. 测试方法

NTB Reserved Memory Size默认设置为512MB，读/写均为顺序读/写，

1. CPU开销

由于DPDK驱动使用CPU Core是独占式，目前的测试均独占式的使用一个CPU Core，CPU利用率100%。

1. 测试结果
2. 针对单核下开启/关闭GNU -O3编译优化进行测试

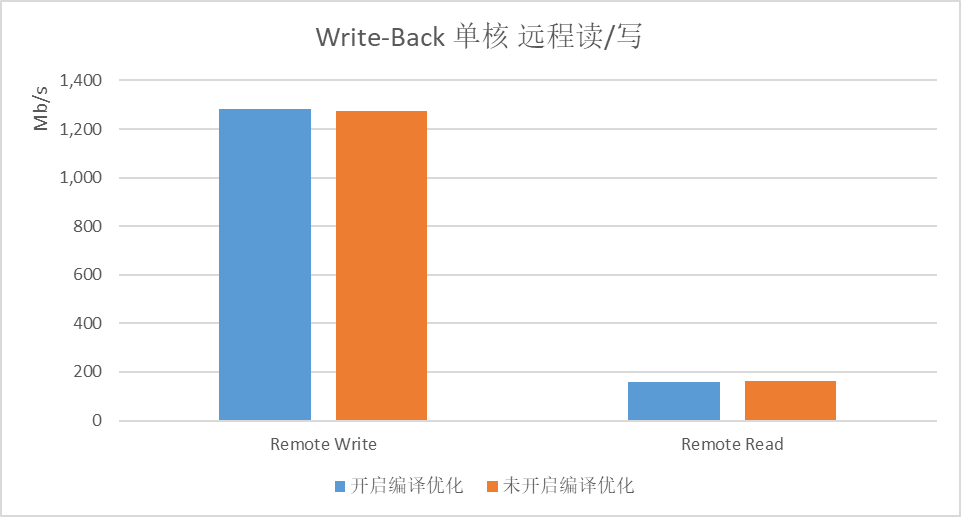


图5.1 Write-Back模式下单核Remote Write/Read速率

图5.2 Write-Back模式下单核LocalWrite/Read速率

**测试结果**：开启GNU -O3编译优化对远程读/写性能没有提升，单核RW速率1.2Gb/s，单核RR速率160Mb/s，但优化了读/写本地内存的速率，本地内存单核读/写速率约500Gb/s左右。之后的测试默认开启编译优化。

1. 针对单核下Memcpy函数R/W内存长度不同的情况进行测试

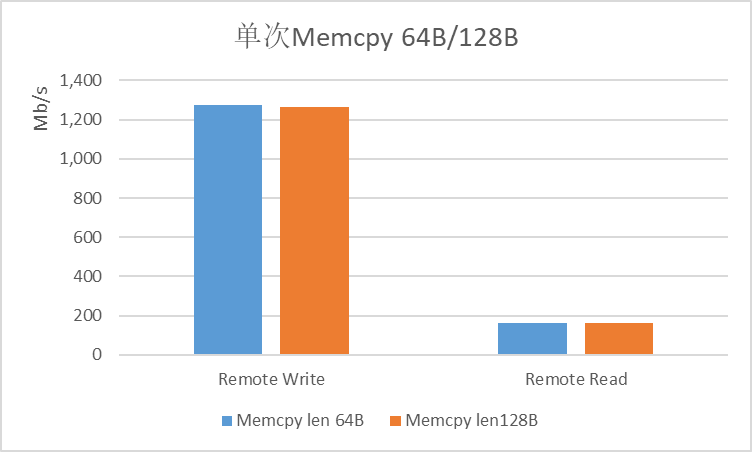


图5.3 Write-Back模式下Memcpy长度对速率影响

测试结果：改变Memcpy函数的单次读/写长度对性能没有提升。之后的测试默认使用64B的单次读/写长度。

1. 针对多核情况下内存使用Write-Back/Write-Combining模式的R/W性能进行测试

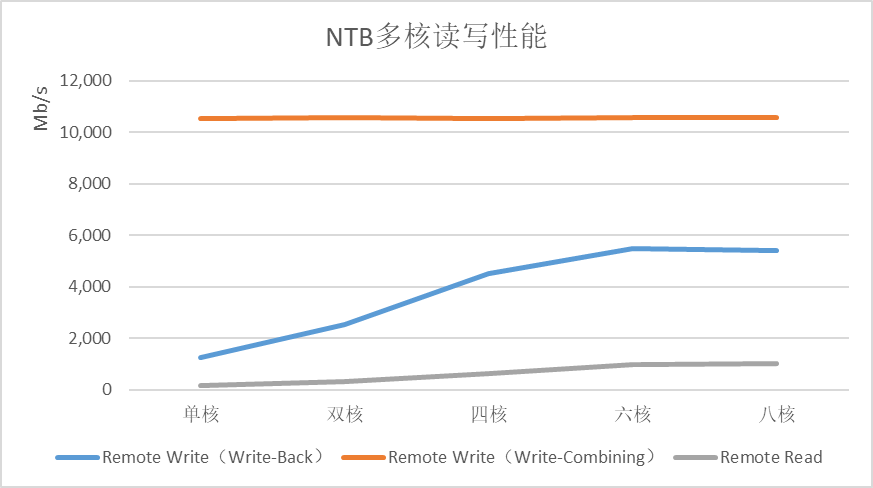


图5.4 NTB多核读写性能测试

**测试结果**：Write-Back模式中，RW、RR吞吐量均在6核左右达到最大，RW速率为5Gb/s，RR速率为1Gb/s。开启Write-Combining模式之后，RW速率在单核时即可达到最大10Gb/s，RR速率在6核时达到最大1Gb/s，与Write-Back下速率相同，图线重合。

**总结：**在开启O3编译优化的情况下，Memcpy长度不影响读写性能，内存区域开启Write-Combining模式后，Remote Write速率在单核时即可达到最大10.2Gb/s，Remote Read速率在6核时达到最大1.1Gb/s，单核170Mb/s。

#### 5.3.2 NTB随机写入性能

根据5.3.1节的测试可知，NTB提供的最大顺序Remote Write性能在10.3Gb/s，顺序Remote Read性能在1.2Gb/s。由于其顺序Remote Read性能远小于顺序Remote Write，并且我们设计NDTP协议的时候避免了使用Remote Read操作，本节测试随机Remote Write性能，没有再对随机Remote Read进行测试。

下图为单核情况下对NTB进行随机Remote Write测试的情况：

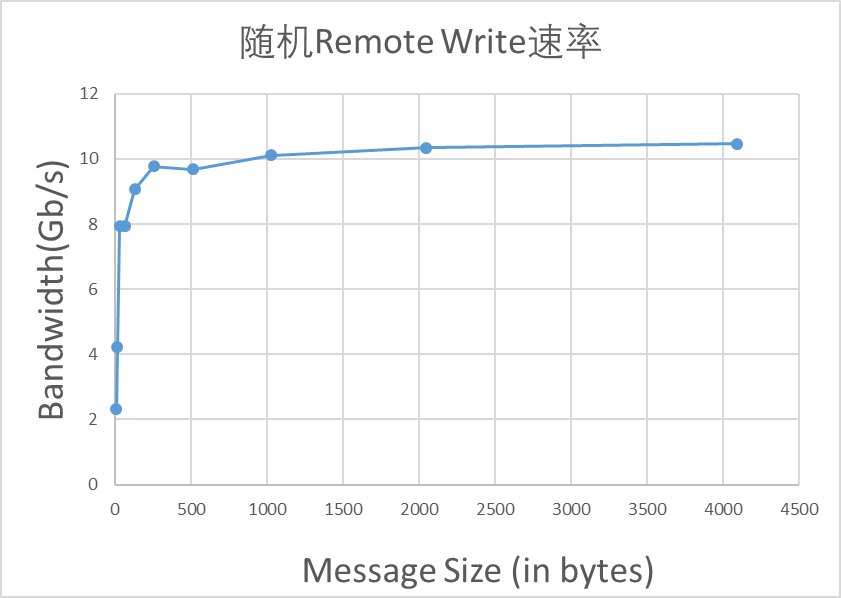


图5.5 NTB随机Remote Write速率

**测试结果**：在Message Size大小为8B时，Bandwidth为2.3Gb/s。在Message Size增加到32B之前，Bandwidth呈线性增长趋势，在Message Size大小为32B时，Bandwidth为7.9Gb/s。之后随着Message Size的增长，Bandwidth呈缓慢增长趋势，在Message Size为256B时，Bandwidth为9.77Gb/s；4096B时，Bandwidth为10.3Gb/s，基本等于顺序读写速率。

#### 5.3.3 NDTP MGT性能测试

构建协议栈后涉及的开销主要有包头开销和从内存池中申请、释放内存节点，以及将内存节点放入Sublink SQ的开销。内存节点的大小由NDTP MGT进程启动时设置。在传输相同数据量情况下，内存节点越大，申请、释放内存节点的操作越少，性能相对就会越高提高。

本节针对MemPool中内存节点MemNode的不同的大小，进行文件传输测试，来评估本文设计的MGT协议栈的性能。

**实验场景**：Server A、Server B均运行一个NDTP管理进程和一个用户进程。

**CPU开销**：由于DPDK驱动使用CPU Core是独占式，目前的测试NDTP进程和应用程序进程均独占式的使用一个CPU Core，CPU利用率均为100%。

下图为不同内存节点大小下的传输速率：

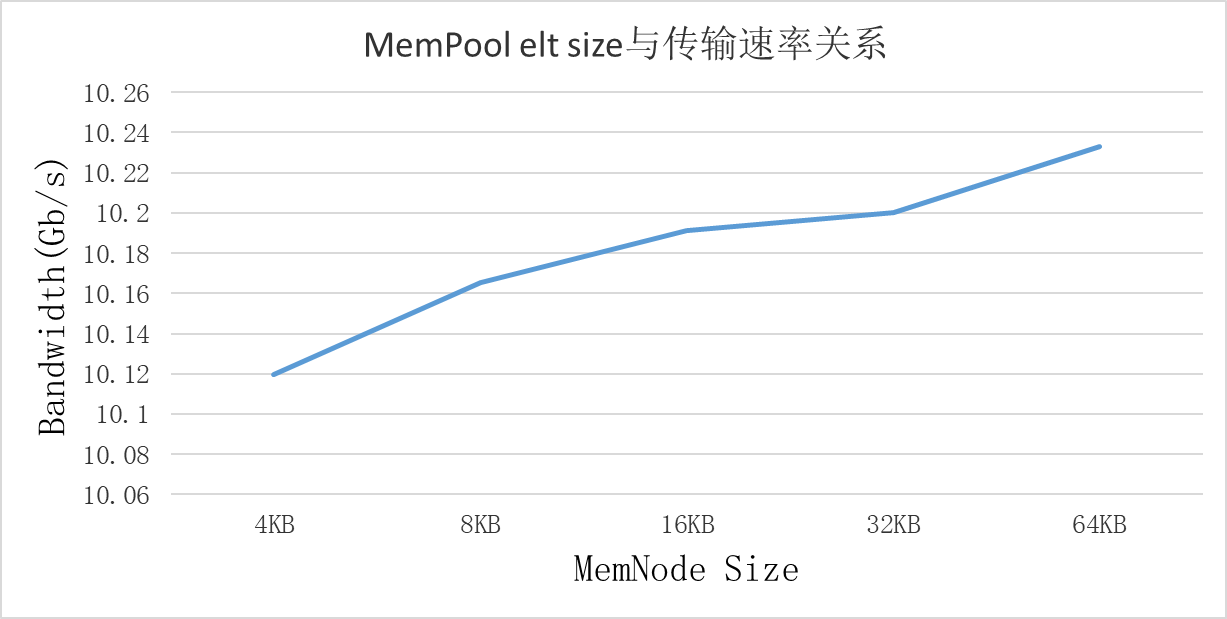


图5.6 MemNode Size与传输速率关系图

**总结：**当MemPool的内存节点MemNode大小从4KB增加到64KB时，传输速率从10.12Gb/s上升至10.23Gb/s，提升约1%。本协议传输效率为（1-首部长度/最大包长）=96.875%。选择4KB作为MemPool节点默认大小时，Remote Write数据传输效率为9.8Gb/s。MemNode Size的增加对传输性能提升不大，因此NDTP默认选择4KB作为MemNode的大小。

#### 5.3.4 传输时延对比测试

本节对TCP、RDMA、NTB三种传输方式的时延进行测试和比较。由于这三种传输方式之间尚没有一个可以通用的测试程序，因此我们选用其各自常用的测试程序进行测试。以太网卡信息、RDMA网卡信息见5.1实验环境介绍。

TCP的传输时延使用NetPerf网络性能测试工具进行测试，网卡设置均采用默认设置，测试模式为Request/Response模式。RDMA的传输时延使用Mellanox官网驱动中的Ping-Pong程序进行测试，传输模式为Wait-For-Notify，传输方式为RDMA\_WRITE/READ。NTP的传输时延是使用本文实现的NDTP提供的Send/Rev接口实现的测试程序进行测试。

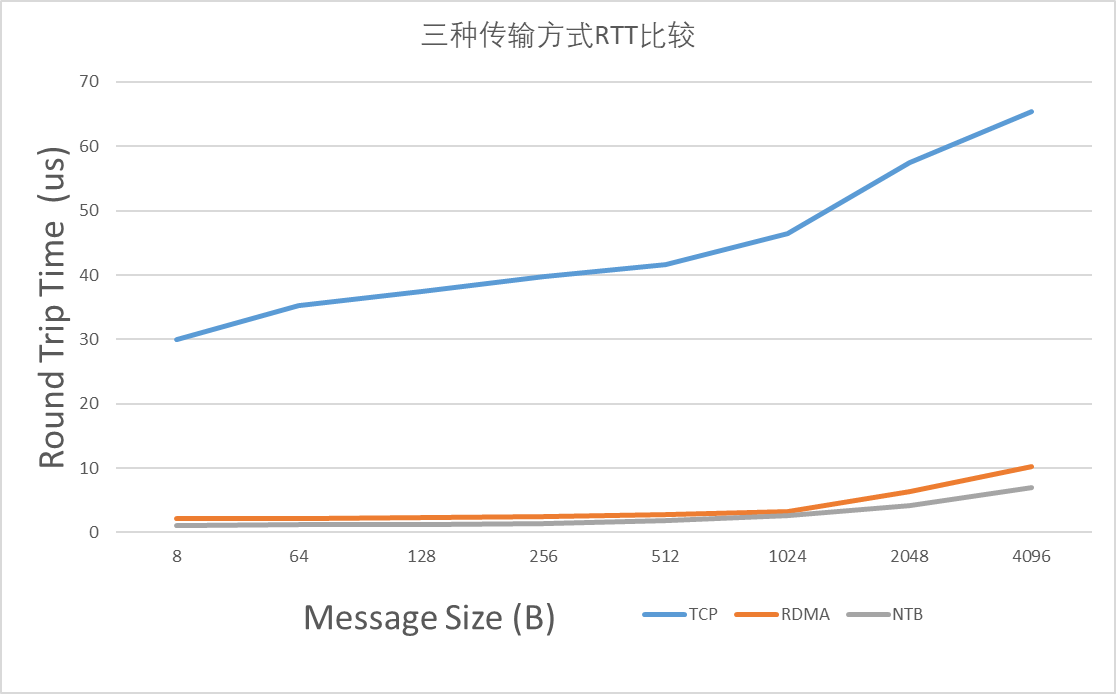


图5.7 TCP/RDMA/NTB传输时延对比

**总结：**同样处于B2B连接情况下，三种传输方式的传输时延均随Message Size的增大而增大，TCP传输方式的RTT远高于其他两种，在Message Size为8B时约为30.5us，4KB时达到65.4us。RDMA往返时延在各个Message Size时均略高于NDTP的传输模式，RDMA的Message Size在8B时为2.12us，而NDTP为1.06us；RDMA 的Message Size在4096B时为10.23us，而NDTP为7.05us。综上所述，在任意Message Size下，NDTP的传输时延均略低于RDMA方式。

## 第六章 总结与展望

### 6.1 论文工作总结

对目前应用于机架内互连的基于PCIe协议的NTB传输方式，论文分析了现有的NTB设备驱动功能限制以及性能瓶颈，实现了一个基于DPDK的NTB用户态驱动程序，在提升了NTB传输性能的同时丰富了驱动程序提供的接口。

随后基于实现的NTB用户态驱动，以及现有PCIe总线协议所能提供的底层网络环境，本文总结并实现了了NDTP传输协议需要提供的功能，实现了多物理机共享、多用户共享、流量控制、连接管理等功能。

本文设计了基于NTB的传输协议NDTP，以及管理NDTP协议运行的NDTP MGT管理程序的整体架构。随后实现了Shared Memory MGT、NBT MGT等相关的管理模块，设计并介绍了其运行逻辑。以及相关的消息首部结构设计、数据结构等，，使NDTP协议能够提供类似于传统网络中传输层的功能。

随后进行对NTB的用户态驱动性能以及NDTP协议性能进行了性能测试，测试了NTB的纯硬件性能以及实现了NDTP协议之后的性能，分析了NDTP传输协议带来的性能损耗。同时使用相关网络测试工具，将NDTP的传输时延与TCP、RDMA等传输方式进行对比，证明了本文设计的NDTP传输协议与其他传输方式相比在传输时延方面有一定的优势。

### 6.2 未来工作展望

对于本文实现的基于NTB的传输协议NDTP来说，仍然有许多可以改进的地方。

首先，NDTP来说虽然考虑了传输优先级相关的设定，例如通过不同的Sublink表示优先级，但优先级并不是由NDTP协议自动分配的，而需要由用户进程指定。同时并没有设计一个合理的优先级调度算法，目前仅仅是从优先级高的发送队列开始发送，这样的设备分配相对来说不够公平。

其次，对于NDTP中运输连接的状态管理尚不完善，在物理设备层面，倘若发生PCIe线路的物理连接断开，目前尚没有一个实时的故障检测以及数据包重传机制。对于NDTP 的Socket连接状态管理不够完善，对于一些复杂网络下的异常情况可能暂时没有处理机制。考虑更具体的应用场景下现有NDTP协议是否还有优化空间，这也是我们之后可以去思考和扩展的方向。

最后，由于为了减少数据在在内存的拷贝次数，我们使用共享内存的方式与用户进程进行数据交换。对于某个Sublink来说，其中的多个连接共享同一个发送队列Sublink SQ，虽然这是一个无锁队列，但当并发数较多的时候仍然会影响性能。因此考虑并发数较多时的性能优化也是未来的工作之一。

## 参考文献

[1] R. Budruk, D. Anderson, and T. Shanley. 2004. PCI Express System Architecture. Addison-Wesley Professional.

[2] [n. d.]. I/O Consolidation White Paper, NextIO, Inc. <http://www.nextio>. com/resources/files/wp-nextio-consolidation.pdf. ([n. d.]). Accessed June 4, 2012.

[3] [n. d.]. Xsigo Systems, Inc. http://en.wikipedia.org/wiki/XsigoSystems.([n. d.]). accessed in July 2017.

[4] 2008. Multi-Root I/O Virtualization and Sharing 1.0 Specification, PCI-SIG Standard. (2008).

[5] A3Cube Inc. [n. d.]. RONNIEE Express Fabric - The In Memory Network . http://www.a3cube-inc.com/ronniee-express.html. ([n. d.]).

[6] William Tu and Mark Lee and Tzi-cker Chiueh. 2013. Secure i/o device sharing among virtual machines on multiple hosts. In Proceedings of ACM International Symposium on Computer Architecture. ACM, 108–119.

[7] J. Byrne, J. Chang, K.T. Lim, L. Ramirez, and P. Ranganathan. 2011. Power-Efficient Networking for Balanced System Designs: Early Experiences with PCIe. In Proceedings of the 4th Workshop on Power-Aware Computing and Systems. ACM, 3.

[8] V. Krishnan. 2007. Towards an Integrated IO and Clustering Solution Using PCI Express. In Proceedings of 2007 IEEE International Conference onCluster Computing. IEEE, 259–266.

[9] William Tu and Tzi-cker Chiueh and Mark Lee. 2014. Marlin: a memorybased rack area network. In Proceedings of the tenth ACM/IEEE symposium on Architectures for Networking and Communications Systems. ACM, 125–136.

[10] 2008. Multi-Root I/O Virtualization and Sharing 1.0 Specification, PCI-SIG Standard. (2008).

[11] R. Budruk, D. Anderson, and T. Shanley. 2004. PCI Express System Architecture. Addison-Wesley Professional.

[12] D. Mayhew and V. Krishnan. 2003. PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects. In Proceedings of 11th Symposium on High Performance Interconnects. IEEE, 21–29.

[13] [n. d.]. PCI Express System Interconnect Software Architecture for x86-based Systems. http://www.idt.com. ([n. d.]). K. Kong, September 2008.

[14] V. Krishnan. 2008. Evaluation of an integrated PCI express I/O expansion and clustering fabric. In Proceedings of 16th IEEE Symposium on High Performance Interconnects (HOTI). 93–100.

[15] D.D. Riley. 2012. System and Method for Multi-Host Sharing of a Single-Host Device. (May 8 2012). US Patent 8,176,204.

[16] J. Regula. 2004. Using Non-Transparent Bridging in PCI Express Systems.PLX Technology, Inc (2004).

[17] STMicroelectronics. [n. d.]. iATU Operations, SPEAr1340 architecture and functionality reference manual. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/referencemanual/DM00024168.pdf. ([n. d.]).

[18] Intel® Xeon® Processor C5500/C3500 Series Datasheet-Volume 1

[19]William Cheng-Chun Tu, Tzi-cker Chiueh: Seamless Fail-over for PCIe Switched Networks. SYSTOR 2018: 101-111

[20]Anuj Kalia, Michael Kaminsky, David G. Andersen: FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. OSDI 2016: 185-201

[21] Ihab Bishara, Data Transfer Between WinNT and Linux Over a Non-Transparent Bridge. COMPUTER ENGINEERING PROGRAM College of Engineering California Polytechnic State University San Luis Obispo

[22] Daniel Münch, Michael Paulitsch, Andreas Herkersdorf:IOMPU: Spatial Separation for Hardware-Based I/O Virtualization for Mixed-Criticality Embedded Real-Time Systems Using Non-transparent Bridges. HPCC/CSS/ICESS 2015: 1037-1044

[23] Cheng-Chun Tu, Chao-Tang Lee, Tzi-cker Chiueh:Marlin: a memory-based rack area network. ANCS 2014: 125-136

[24] Daniel Münch, Michael Paulitsch, Oliver Hanka, Andreas Herkersdorf:MPIOV: scaling hardware-based I/O virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (multi-core) multi-processor systems. DATE 2015: 579-584

[25]Rui Hou, Tao Jiang, Liuhang Zhang, Pengfei Qi, Jianbo Dong, Haibin Wang, Xiongli Gu, Shujie Zhang:Cost effective data center servers. HPCA 2013: 179-187

[26] Daniel Münch, Michael Paulitsch, Oliver Hanka, Andreas Herkersdorf:MPIOV: scaling hardware-based I/O virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (multi-core) multi-processor systems. DATE 2015: 579-584

[27]Lee Mohrmann National Instruments, Austin, Texas USA; Creating multicomputer test systems using PCI and PCI Express. IEEE 2009 Sept. 14-17

[28] William Tu and Mark Lee and Tzi-cker Chiueh. 2013. Secure i/o device sharing among virtual machines on multiple hosts. In Proceedings of ACM International Symposium on Computer Architecture. ACM, 108–119.

[29] PXI Systems Alliance “PXImc Specification”, unpublished

[30] Peripheral Component Interconnect Special Interest Group, “PCI Express® External Cabling 1.0 Specification”, <http://www.pcisig.com/specifications/pciexpress/pcie_cabling1.0/>

[30] Data Plane Development Kit 19.08 “Docs » Rawdev Drivers » 5. NTB Rawdev Driver”, http://doc.dpdk.org/guides/rawdevs/ntb.html

## 致谢

光阴似箭，日月如梭，两年半的研究生生涯即将划上一个句号，而于我的人生来说却仅仅只是一个逗号，我将面对新的征程的开始。本研究及论文是在我的导师梁瑾老师的亲切关怀和耐心的指导下完成的。伟人、名人固然为我所崇拜，可是我更迫切地想要把我的敬意献给我的导师———梁瑾老师。也许我不是您最出色的学生，在许多地方劳您费心，但您却是我所最尊敬的老师。您治学严谨，学识渊博，视野广阔，思想深刻，金益求精，您用心为我营造一种良好的学术氛围，让我的论文更加的严谨。同时，我还要感谢一下实验室的同学们，在毕业设计中遇到的许多问题，如果没有你们的帮助，我可能需要耗费大量的时间去绕开一些思维定式。

至此论文完成之际，我的心情着实难以平静，从开始选择课题到论文的顺利答辩，有无数可敬的师长、朋友给了我很多的帮助，在这里请您接受我诚挚的谢意! 最后，再次对那些在论文完成过程中，关心、帮助我的同学和朋友们表示衷心地感谢！

**复旦大学**

**学位论文独创性声明**

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。论文中除特别标注的内容外，不包含任何其他个人或机构已经发表或撰写过的研究成果。对本研究做出重要贡献的个人和集体，均已在论文中作了明确的声明并表示了谢意。本声明的法律结果由本人承担。

作者签名： 日期：

**复旦大学**

**学位论文使用授权声明**

本人完全了解复旦大学有关收藏和利用博士、硕士学位论文的规定，即：学校有权收藏、使用并向国家有关部门或机构送交论文的印刷本和电子版本；允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。涉密学位论文在解密后遵守此规定。

作者签名： 　 导师签名： 日期：