学校代码：10246
学　　号：16110240004

復旦大學

博 士 学 位 论 文

（学术学位）

基于内核旁路网络的系统性能增强和协议栈设计研究

Research on System Performance Enhancement and Protocol Stack Design with Kernel Bypass Network

院　　系：计算机科学技术学院
专　　业：计算机应用技术
姓　　名：黄一博
指 导 教 师：吴杰　研究员
完 成 日 期：2021 年 10 月 10 日

指导小组成员

张世永　教　授
钟亦平　教　授
吴　杰　研究员
吕智慧　教　授
徐　扬　教　授
吴承荣　副教授
曾剑平　副教授

目 录

插图目录

表格目录

摘 要

在当今数据规模呈指数级增长和数据驱动的趋势下，分布式系统已经成为数据中心支持高质量互联网服务的重要基础。分布式系统通过多节点之间数据中心网络驱动的资源聚合池化方式，为上层应用提供计算与存储服务，可以分为吞吐/内存敏感、时延/CPU 敏感两种系统类型：吞吐/内存敏感系统提供大规模数据的访存和可用性保证，依赖高带宽、低时延的数据传输，内存开销大，如 NoSQL（Not only SQL）系统；而时延/CPU 敏感系统提供关键数据的完整性和一致性保证，依赖频繁的点到点通信驱动分布式共识，CPU 开销大，如区块链系统。这对网络的性能提出了更高的要求。许多数据中心分布式系统的设计与实现默认假设网络传输比本地内存处理速度更慢。

近年来，40Gbps、100Gbps 甚至 400Gbps 的新型高速网络在数据中心的规模化部署打破了上述假设。许多新型网络又称为内核旁路网络，如 RDMA（Remote Direct Memory Access）和 PCIe（Peripheral Component Interconnect Express）互连，通过旁路操作系统内核和协议栈硬件卸载，允许本地网卡在没有远端 CPU 的干预下直接访问远端机器的内存，从而提供高吞吐、超低时延的网络性能。但是，这些性能优势依赖于新的通信原语，与传统分布式系统的通信抽象（如网络套接字）不匹配，需要对分布式系统通信层的重新设计才能实现整体性能提升。因此，如何充分利用内核旁路网络的特性和通信原语获得高性能、资源高效、兼容扩展的高质量服务，是数据中心规模的分布式系统面临的关键问题和挑战。

本文以优化分布式系统的性能、资源开销和兼容扩展性为研究目标，旨在探索利用内核旁路网络技术进行分布式系统性能增强和高性能协议栈设计。本文首先引入自上而下的系统适配通信原语的设计思想，从网络与系统协同的角度，提出利用商用 RDMA 网络对系统通信层的重新设计，在满足资源高效的同时增强吞吐/内存敏感系统和时延/CPU 敏感系统的性能。然后引入自下而上的通信原语适配系统的设计思想，为了解决商用 RDMA 网络在机架内通信场景下 PCIe 协议与网络协议的转译开销问题，从软硬件协同的角度，提出基于 PCIe 非透明桥（Non-Transparent Bridge，NTB）新硬件的轻量级高性能用户态协议栈，可以在系统不变动或最小化系统变动的情况下增强不同系统类型的数据传输性能。主要的创新性贡献如下：

（1） 从网络和吞吐/内存开销敏感系统协同设计的角度，提出了一种 RDMA 增强的高性能、内存高效、高可用的文档数据模型，并实现了对应的文档 NoSQL 系统 RMongo。为了传输链路的容错性，本文设计了 RDMA 上下文检测算法，以决定两个系统节点之间使用 RDMA 或传统 TCP/IP 通信；设计了负载感知的 RDMA 缓冲区注册机制，以实现内存高效的 RDMA 通道内存区域管理；利用基于内存语义的单边 RDMA 操作原语扩展面向文档数据模型的增删改查操作语义，提升了文档 NoSQL 的端到端性能；为了文档数据模型的高可用，结合单边 RDMA 原语和异步完成事件通道重新设计了多切片节点之间操作日志同步协议。实验表明，相比传统的基于 Linux TCP 的文档 NoSQL，RMongo 可提升端到端数据操作的吞吐量达 30%。

（2） 从网络和时延/CPU 开销敏感系统协同设计的角度，提出了一种 RDMA 驱动的高性能、CPU 高效、可扩展的分布式共识协议，并在此基础上实现了对应的区块链系统 BoR。本文利用单边 RDMA 内存操作原语和基于 RDMA 完成事件通道的异步事件模型，重新设计驱动分布式共识的点到点通信，绕过服务端 CPU，使得 BoR 既实现了更低的共识同步时延，又最小化了通信带来的 CPU 开销；利用混合 RDMA 原语和共享接收队列设计了新节点引导协议，BoR 降低了新节点初始化同步历史区块数据的时延，提高了共识网络的扩展性。实验表明，在不同规模的工作负载下，相比于基于 Linux TCP 的区块链系统，BoR 的新节点共识过程降低了 CPU 开销达 26.4%，降低了区块同步时延达 20.2%。

（3） 从软硬件协同设计的角度，提出了一种基于 PCIe 互连的、控制平面与数据平面分离的高性能用户态协议栈 NTSocks。考虑到新硬件 PCIe 非透明桥互连比 RDMA 少了 PCIe 协议与网络协议之间的转译开销，实现了 2.3~5.6 倍的通信性能加速，且组网拓扑更简单，本文设计了面向机架规模通信的基于 PCIe NTB 新硬件的轻量级协议栈。通过设计兼容的类似套接字的网络功能抽象，NTSocks 以用户态运行时库的形式解决了原生 PCIe NTB 抽象不匹配的问题。为了数据平面的多核扩展性，NTSocks 基于并行化思想提出了 CPU 核心驱动的数据平面模型。为了公平高效的资源共享，NTSocks 设计了面向多租户的性能隔离机制。尽管 PCIe NTB 最初是为跨 PCIe 域间的设备通信而设计，NTSocks 展示了一个灵活的用户级间接层，可以在提供通用网络功能的同时实现接近裸机 PCIe NTB 的网络性能。实验表明，相比 Linux TCP 和 RDMA 套接字，NTSocks 降低微基准测试时延分别到 1/32 和 1/4.4，降低键值存储系统的端到端时延分别到 1/24.5 和 1/1.58，将 Nginx HTTP 文件服务响应时延降低至 1/6.7。

关键字：内核旁路网络；分布式系统；网络协议栈；RDMA；PCIe 互连；PCIe 非透明桥；通信架构

中图分类号：O413.1

Abstract

With the exponential growth of data scale and the trend of data-driven paradigm today, distributed systems have become an significant foundation for data centers to support high-quality Internet services. Distributed systems provide computing and storage services for upper-layer applications through the resource aggregation and pooling approach driven by data center network among multiple nodes. It can be divided into two types of systems: throughput/memory-sensitive and latency/CPU-sensitive. Throughput/memory-sensitive systems (e.g., NoSQL systems) rely on high-bandwidth and low-latency data transmission to provide large-scale data access and availability guarantee, which generally result in large memory overhead. Latency/CPU-sensitive systems (e.g., blockchains systems) rely on frequent point-to-point communications to drive distributed consensus to ensure data integrity and strong consistency, which generally introduce large amounts of CPU overhead. Thus, more tighter requirements on network performance are strongly demanded by distributed systems. Many distributed systems in data centers are designed and implemented with a default assumption that network transmission is relatively slower than local memory access speed.

In recent years, the large-scale deployment of modern high-speed networks of 40Gbps, 100Gbps and even 400Gbps in data centers has broken the above assumption. Many modern networks are also termed kernel-bypassing networks like commercial RDMA (Remote Direct Memory Access) and lightweight PCIe (Peripheral Component Interconnect Express) fabric. By bypassing the operating system (OS) kernel and offloading network protocol stack into dedicated hardware, they allow the local network interface card (NIC) to directly access the memory of remote machines without any involvement of remote host CPU, thereby achieving high-throughput and ultra-low-latency network performance. However, these performance advantages rely on brand-new communication primitives, which don't match the communication abstractions (e.g., network socket) of traditional distributed systems. The transport layer of distributed systems is required to be co-designed closely with the characteristics of kernel-bypassing network to boost overall system performance. Therefore, it is a critical problem and challenge for datacenter-scale distributed systems to make full use of the characteristics of kernel-bypassing network and new communication primitives to obtain high-performance, resource-efficient, compatible and scalable high-quality services.

With the research goal of optimizing the performance, resource efficiency, compatibility and scalability of distributed systems, the paper aims to explore how to leverage kernel-bypassing network technology for system performance enhancement and lightweight protocol stack design. First, the paper introduces the design idea of top-down system-to-primitive adaptation. From the perspective of network-system co-design, re-designing system transport layer with commercial RDMA network is proposed to achieve resource efficiency and extremely enhance the performance of the above two system types. Then, the paper introduces the design idea of bottom-up primitive-to-system adap-

tation. To solve the problem of translation overhead between PCIe protocol and network protocol in commercial RDMA for in-rack network, from the perspective of software-hardware co-design, the paper proposes a high-performance user-space protocol stack based on modern PCIe non-transparent bridge (NTB) hardware from scratch. The lightweight protocol stack can compatibly enhance the data transfer performance of different system types without any or with minimal modification. The critical innovative contributions are as follows:

(1) From the perspective of the co-design of network and throughput/memory-sensitive systems, the paper proposes an RDMA-enhanced high-performance, memory-efficient, and highly available document data model, and implements the corresponding document NoSQL system, RMongo. For the fault tolerance of the transport links, an RDMA context detection algorithm is designed to determine the selection of RDMA or kernel TCP/IP links between two nodes. A load-aware RDMA buffer registration mechanism is designed to achieve memory-efficient RDMA memory region management. The operation semantics (i.e., insert, delete, update and query) of the document-oriented data model is extended by memory-semantic one-sided RDMA primitives, which improves the end-to-end performance of document NoSQL systems. For the high availability of the document data model, the operation log synchronization protocol between multiple shard nodes is redesigned by combining one-sided RDMA primitives and asynchronous completion event channel. Experimental results show that, RMongo can improve the end-to-end throughput of document data model by 30% compared to traditional TCP-based document NoSQL.

(2) From the perspective of the co-design of network and latency/CPU-sensitive systems, the paper proposes an RDMA-driven high-performance, CPU-efficient, and scalable distributed consensus protocol, and implements the corresponding permissioned blockchain system, BoR. By exploiting one-sided RDMA primitives and RDMA completion-based asynchronous event model, BoR redesigns the point-to-point communication that drives distributed consensus protocol and bypasses server CPU, which achieves lower block/transaction synchronization latency and minimizes CPU overhead. BoR leverages hybrid RDMA primitives and shared receive queues to design a new node bootstrapping protocol, which reduces the delay of initial blocks synchronization for new nodes and improves the scalability of the consensus network. Experiments show that, compared to traditional TCP-based distributed consensus protocol, BoR reduces CPU overhead by 26.4% and block synchronization latency by 20.2% under various-scale workloads.

(3) From the perspective of software-hardware co-design, the paper proposes NTSocks, a lightweight user-level protocol stack over PCIe fabric, which seperates the data and control plane. Modern PCIe NTB fabric eliminates the translation overhead between the PCIe and network protocol and has a flatter networking topology, which can achieve more lower transfer latency by 2.3 to 5.6 times than commercial RDMA. So, the paper proposes a lightweight protocol stack based on modern PCIe NTB hardware for in-rack communication. To solve the issue of native PCIe NTB abstraction mismatch, we propose compatible socket-like network abstraction with a user-level runtime library. For dataplane multi-core scalability, NTSocks adopts the idea of parallelization to propose a CPU core-driven dataplane model. For fair and efficient resource sharing, NTSocks proposes a multi-tenant performance isolation mechanism. Despite the fact that PCIe NTB is originally designed

for device communication across PCIe domains, NTSocks shows that a flexible user-level indirection can deliver performance close to bare-metal NTB while providing common network stack features. Experiments show that, compared to Linux TCP and RDMA sockets, NTSocks reduces the micro-benchmark latency by 32 times and 4.4 times, respectively. With key-value store, NTSocks achieves better latency by up to 24.5 times and 1.58 times than kernel and RDMA socket, respectively. NTSocks reduces the response delay of Nginx HTTP file service by 6.7 times than Linux TCP.

Keywords: Kernel-bypassing network; Distributed system; Network protocol stack; RDMA; PCIe fabric; PCIe Non-Transparent Bridge; Communication architecture

CLC number: O413.1

主要符号对照表

RDMA	远程内存直接访问（Remote Direct Memory Access）
PCIe	外围设备互联（Peripheral Component Interconnect Express）
NTB	非透明桥（Non-Transparent Bridge）
TLP	事务层数据包（Transaction Layer Packet）
TSO	TCP 段卸载（TCP Segment Offload)
RoCE	融合以太网 RDMA（RDMA over Converged Ethernet）
RC	可靠连接（Reliable Connection）
UC	不可靠连接（Unreliable Connection）
UD	不可靠数据报（Unreliable Datagram）
MTU	最大传输单元（Maximum Transmission Unit）
QP	队列对（Queue Pair）
CQ	完成队列（Completion Queue）
WQ	工作队列（Work Queue）
DPDK	数据面开发包（Data Plane Development Kit）
MMIO	内存映射 I/O（Memory-Mapped I/O）
UIO	用户空间 I/O（Userspace IO）
PFC	优先级流量控制（Priority-based Flow Control）
oplogs	操作日志（Operation Log）
PMD	轮询模式驱动（Poll-Mode Driver）
HOL blocking	线头阻塞（Head-Of-Line blocking）
SoC	片上系统（System-on-Chip）
CNP	拥塞通知数据包（Congestion Notification Packet）
MW	内存窗口（Memory Window）
BAR	基地址寄存器（Base Address Register）
WC	写入合并（Write Combining）
DDIO	数据直接访存技术（Data Direct I/O）
FCT	流完成时间（Flow Completion Time）
QoS	服务质量（Quality of Service）
DPU	数据处理单元（Data Processor Unit）
PCC	可编程拥塞控制（Programmable Congestion Control）

第 1 章　绪论

1.1　研究背景和意义

近些年，随着互联网规模的快速发展，数据处理规模呈指数级增长，数据中心规模的分布式系统已经成为支撑高质量互联网服务的重要基础。数据驱动的分布式系统依赖于数据中心网络驱动的多节点协同计算与存储，这需要多节点之间密集的数据传输。例如，抖音短视频有超过 1.5 亿的日活用户量，支撑短视频推荐的后端分布式系统（如 PolarFS[1]）每天需要处理超过 294TB 的视频流数据，同时还要提供超低的端到端时延以保证体验质量（Quality of Experience，QoE）[2]。因此，高性能的数据传输对于保障分布式系统的服务质量尤为重要。

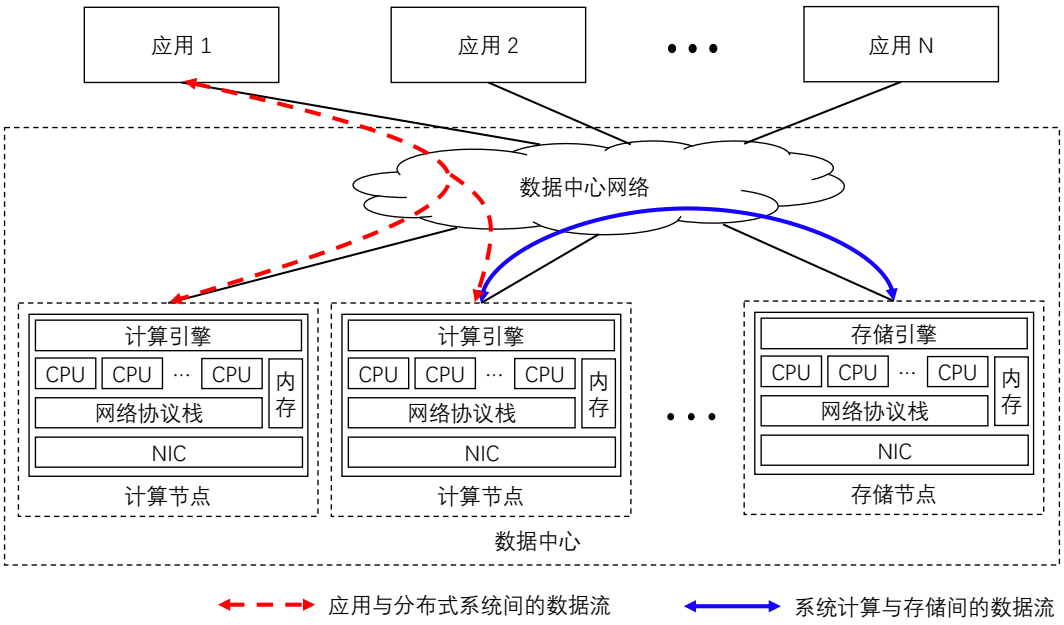


图 1-1　计算与存储分离的数据中心设计

为了解耦计算和存储以提高灵活性，许多数据中心系统开始从以计算为中心的传统设计过渡为计算与存储分离的架构，使得网络成为了连接计算与存储的“桥梁”[3, 4]，如图 1-1 所示。同时，许多新型计算（如 GPU、TPU、FPGA）和存储（如非易失性内存）硬件加速器在数据中心部署以增强数据驱动处理能力[5]，如 Intel 3D-XPoint 作为非易失性内存的典型硬件实现，可以提供 10 微秒左右的读写时延[1]，这进一步使得系统内核干预的传统网络栈逐渐变成分布式系统的一个潜在的性能瓶颈。在这样的背景下，分布式系统和网络协议栈迎来了巨大挑战，具体包括：

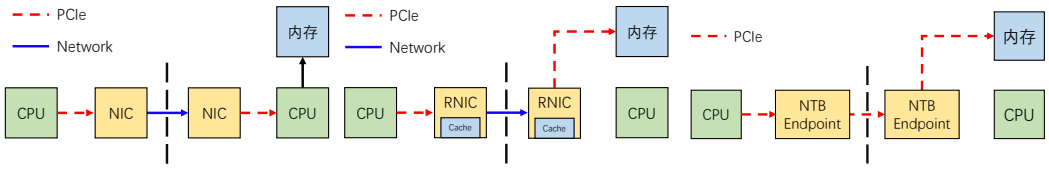
- **吞吐时延要求：**数据处理规模的增长要求数据传输性能的提升。分布式系统与协议栈的性能要求主要包括带宽与时延要求。一方面，在服务于大规模互联网应用的分布式

系统中，包括频繁的数据采集预处理、中间数据同步在内的访存操作，需要分布式系统提供很高的带宽。如阿里巴巴电子商务数据库在峰值阶段需要处理每秒 54400 个订单的吞吐量[4]。另一方面，互联网服务对于端到端的响应时间有更严苛的要求，如 Google 搜索引擎需要在用户输入查询内容的时候，在数十毫秒时延内推理预测出潜在的搜索关键词[6]。

- **低开销要求：**大规模数据处理需要消耗大量的计算和内存资源，这要求分布式系统与协议栈尽可能降低资源开销，提升计算和内存资源的复用率，避免资源争抢而导致的性能降低。例如，Google Memorystore 云服务通过将数据放置在内存中，提供容量高达 5 TB 和亚毫秒级数据访问的内存数据缓存系统[7]。微软 Azure 推出区块链即服务，通过消耗许多计算资源的分布式共识算法来对外提供数据存储的完整性和可靠性服务[8]。

- **扩展性要求：**在分布式系统与协议栈中，海量数据的高并发处理要求数据路径具备面向多核、多机的横向扩展能力。高并发请求处理的性能缩放能力显著地影响互联网服务的可用性。例如，MongoDB 通过数据分片将数据的多个副本横向扩展到多个分片节点上，以提高数据的可用性[9]。2018 年亚马逊 S3（Simple Storage Service）经历了持续近 4 个小时的系统宕机，直接影响了亚马逊弹性云计算、块存储等服务，并严重影响了运行在亚马逊云上的上千个网站服务[4]。同时，在高并发流量负载下，协议栈的数据平面要维持可扩展缩放的数据转发能力。

然而，主流的传统内核态网络协议栈（如 TCP/IP）通常需要几百微秒的传输时延[10]，且单内核线程所达到的吞吐仅为数十个 Gbps[4]，很难满足分布式系统的上述要求。尽管普通以太网卡的带宽已经从 10Gbps 发展到 40Gbps 甚至 100Gbps，传统 TCP/IP 网络栈仍然通过操作系统内核封装数据平面和控制平面的实现，在关键数据路径上引入了很大开销[11]。例如，一个基于传统 TCP 的数据包单向传输过程，包含了用户空间、内存空间和网卡之间至少 4 次内存拷贝，2 次中断处理，应用进程与内核线程之间的 2 次上下文切换开销，套接字缓冲区的锁竞争，且协议栈对数据包头部的封装和解析引入了不可忽视的 CPU 开销。分布式系统中海量数据规模的传输进一步使得传统网络协议栈引入了巨大的 CPU 和内存开销，减少了可用的 CPU 和内存资源，这对运行在分布式系统上的计算、存储任务产生了不可容忍的资源竞争，从而降低了系统整体性能。



(a) 传统以太网　　(b) RDMA　　(c) PCIe NTB

图 1-2　传统以太网、RDMA 和 PCIe NTB 的数据路径比较

近年来，新型内核旁路网络的出现与快速发展为分布式系统性能增强和高性能协议栈设计带来了重要发展机遇。一方面，以 RDMA 为代表的商用内核旁路网络已经在许多数据中

表 1-1　内核旁路网络与传统网络的性能比较

类型	设备型号	PCIe 总线	时延 (μs)	带宽 (Gbps)
传统以太网	Intel 以太网卡 X722	GEN 3.0×16	上百微秒	25
RDMA	Mellanox MT28800 ConnectX-5	GEN 3.0×16	~3	100
PCIe NTB	Intel Sky Lake-E NTB Registers	GEN 3.0×16	~1	接近 PCIe 带宽

心部署和应用，通过把整个网络协议栈卸载到网卡硬件，允许本地网卡绕过远端主机的 CPU 直接读写远端的内存[10, 12]。另一方面，一些新硬件如 PCIe 非透明桥（PCIe Non-Transparent Bridge，PCIe NTB）也提供了轻量级内核旁路网络的能力，通过硬件上基于 PCIe 事务层协议（Transaction Layer Protocol）的域间地址转译实现主机之间的内存共享，允许本地用户态进程绕过远端主机的 CPU 直接读写远端内存，消除了 PCIe 协议与网络协议之间的转译，在机架规模通信场景内提供了比商用 RDMA 更低的时延[13, 14]。RDMA 和 PCIe NTB 两种内核旁路网络相比于传统 TCP/IP 网络栈，通过硬件实现的协议栈提供了超低时延、高带宽、内存语义零拷贝、低 CPU 开销、异步通信等优点，如图 1-2 所示。这些特性在分布式系统和协议栈的吞吐时延性能、低开销和扩展性要求中均带来了全新的机遇和挑战。在吞吐时延性能要求方面，内核旁路网络的传输时延和带宽均突破了传统内核态 TCP/IP 网络的性能瓶颈。表 1-1 对比了当前不同内核旁路网络和传统以太网的传输性能差异。相比于传统网络，RDMA 和 PCIe NTB 的传输时延降低了两个数量级，带宽也有了大幅度提升，最新的商用 RDMA 网卡硬件甚至支持 200Gbps 的带宽。在低开销要求方面，传统 TCP/IP 网络数据路径的数据拷贝、锁竞争、中断处理等均限制了资源的可用性。RDMA 和 PCIe NTB 相比于传统以太网有 CPU 绕过、内存语义零拷贝等特性，例如通过协议栈硬件卸载和 CPU 绕过节省 CPU 资源，通过内存语义零拷贝节省内存资源，因而在分布式系统的资源可用性方面更具有竞争力。在扩展性要求方面，相比于传统 TCP/IP 网络，RDMA 通过异步通信模型重叠通信与计算以提高 CPU 效率，从而支撑更高的并发请求。得益于内存语义的消息传输，PCIe NTB 和 RDMA 的单核吞吐甚至可以使总带宽达到饱和，且数据面 CPU 核数的增加可以使总带宽达到饱和状态，这在第 5.6 节实验评估部分有讨论。但与传统以太网卡不同，RDMA 和 PCIe NTB 的硬件资源具有局限性，例如 Mellanox RDMA 在可靠连接模式中依赖于网卡上有限的页表缓存和连接上下文缓存提供高性能[15]，Intel Sky-Lake PCIe NTB 当前仅支持 512MB 的共享内存[13]。内核旁路网络硬件资源的局限性成为分布式系统和协议栈扩展性设计的新挑战。

分布式系统依赖于网络驱动的多节点协同，从而为上层用户应用程序保证服务质量。分布式系统与网络的协同设计、通信协议栈与网络硬件的协同设计对于新型内核旁路网络特性的发挥十分重要。传统分布式系统依赖于内核态网络套接字的数据传输，例如利用 TCP/IP 栈的段卸载（TCP Segment Offload，TSO）特性提升传输吞吐量，而 TCP/IP 栈利用网卡硬件对数据分段的支持来实现 TSO 特性[16]。然而基于传统 TCP/IP 网络的分布式系统不再适应于内核旁路网络，因为内核旁路网络的性能优势依赖于新的网络通信原语抽象和通信模型，与传统的套接字抽象不匹配。传统的系统与协议栈既无法发挥出内核旁路网络的性能优势，也无法屏蔽内核旁路网络的不足。因此，本文选择面向内核旁路网络的系统增强和高性能协议栈设计关键技术展开研究。

1.2　内核旁路网络、系统与协议栈概述

1.2.1　内核旁路网络

相比于传统内核态网络栈，内核旁路网络在数据路径上绕过操作系统内核，从而消除了中断处理、系统调用、多次内存拷贝、锁竞争等关键开销。目前主流的内核旁路网络主要分为两种：（1）协议栈硬件卸载，即网络协议栈卸载到硬件网卡上；（2）用户空间 I/O，即将网络协议栈上移到用户态，如 mTCP[11]。RDMA 和 PCIe NTB 均属于协议栈硬件卸载的方式。

1. RDMA

远程直接内存访问（Remote Direct Memory Access，RDMA）是一种面向高性能网络 I/O 的协议标准，定义了在没有远端主机 CPU 的中断处理下直接访问（即读、写）远端主机内存的规范。RDMA 最初起源于由 IBTA（InfiniBand Trade Association）定义的专用于高性能计算的 Infiniband 工业标准，在 2013 年后逐步受到学术界和工业界的关注。常见的 RDMA 实现主要包括 RoCE（RDMA over Converged Ethernet），InfiniBand（IB），Omni-Path 和 iWARP，RoCE 又分为 RoCEv1 和 RoCEv2，分别运行在以太网二层链路层、以太网四层 UDP 之上。其中，RoCEv2 在以太网数据中心中部署较为广泛，若无特别说明，下文 RDMA 均指 RoCEv2 类型。

RDMA 的传输类型可以分为可靠的连接（Reliable Connection，RC）、不可靠连接（Unreliable Connection，UC）和不可靠数据报（Unreliable Datagram，UD），不同的传输模式对于吞吐/时延性能、传输可靠性、通信原语方面的支持有明显的区别。RC 和 UC 模式均为面向连接的数据传输，每次转发的有效载荷最大可支持 2GB，单核吞吐可以很容易达到线速。而不可靠数据报模式类似于 UDP 不需要建立一对一的连接，支持单播和广播，每次转发的有效载荷最大为网卡的最大传输单元（Maximum Transmission Unit，MTU）大小，单核吞吐很难达到线速。RC 相比于 UC 把可靠传输机制卸载到网卡硬件的网络层，例如 RDMA 操作的完成需要本地网卡收到对端网卡上发来的确认数据包，并通过 *go-back-to-n* 算法[12] 实现丢包重传，硬件实现的可靠性保证使得上层系统的设计不需要关心通信的可靠性。不同的传输模式对于 RDMA 通信原语的支持有所不同，如表 1-2 所示，RC 模式支持所有的通信原语，能充分反映出 RDMA 的性能优势。

表 1-2　RDMA 传输模式与通信原语 Verbs 的关系[17]

RDMA 原语	UD	UC	RC
Send (with immediate)	✓	✓	✓
Receive	✓	✓	✓
RDMA Write (with immediate)	×	✓	✓
RDMA Read	×	×	✓
Atomic	×	×	✓
Max message size	MTU	2GB	2GB

RDMA 以运行时库（即 *ibverbs* 与 *rdmacm*）的形式为上层应用提供两种类型的高性能通信原语：面向消息通道语义的双边原语（即 *Send/Recv Verbs*）、面向内存语义的单边原语（即 *Write/Read/Atomic Verbs*）。这些通信原语抽象的合理使用对于发挥 RDMA 的零拷贝、低 CPU 开销和性能特性至关重要。对于双边 RDMA Send/Recv 原语，与传统套接字类似，需要请求方提供发送内存缓冲区，响应方提供接收内存缓冲区，双方的数据收发均需要 CPU 的干预。对于单边 RDMA Write/Read 原语，由请求方提供发送缓冲区和接收缓冲区，对远程内存的访存操作本质上是请求方硬件网卡直接绕过响应方（即远端）CPU 在发送缓冲区和响应方的接收缓冲区之间同步数据，这种方式天然支持数据零拷贝特性，且响应方的 CPU 开销几乎为零。带有即时数据的 RDMA Write 操作（又称为 *Write with immediate*）和 Atomic 原子操作均是单边 RDMA 操作的变种。在数据传输量较大的场景中，单边 Write/Read 原语相比于双边 Send/Recv 原语有明显的时延和吞吐优势，并且节省内存、CPU 资源开销，在客户端/服务端应用架构中可以有效改进存在计算资源瓶颈的服务端的并发处理能力。

而 RDMA 内存缓冲区管理依赖于内存区域（Memory Region，MR）注册/反注册机制，允许应用程序申请连续的虚拟内存或物理内存空间，提供了安全高效的内存语义操作，却也带来了较大的开销。具体而言，注册 RDMA MR 的应用进程首先通过操作系统锁定指定的内存页，以防止对应的物理内存被操作系统交换出去；然后，应用进程把虚拟地址到物理地址的映射页表同步到 RDMA 网卡页表内，设定内存区域对应的读、写等权限，每个 MR 有对应的远程 key（remote key，rkey）和本地 key（local key，lkey），对 RDMA MR 的远程和本地读写操作分别需要 rkey、lkey；最后在释放 MR 时需要显式地进行反注册操作。因而，RDMA MR 注册/反注册机制是复杂且 CPU 开销较大的，频繁的 RDMA 注册/反注册操作会引入不可忽视的 CPU 和时延开销。

RDMA 基于完成队列（Completion Queue，CQ）的数据传输模型实现异步通信特性，从而实现通信与计算的重叠，提高了 CPU 效率。在 RDMA 可靠传输模式下，每个连接创建一个 Queue Pair（QP）对象，在网卡硬件上包含了发送队列、接收队列两种工作队列（Work Queue，WQ），每个 QP 绑定到一个 CQ 上。数据的收发事件均可以放入 CQ 中，并通过应用进程显式询问或通过异步完成事件通道通知应用进程，应用进程在通信过程中不需要阻塞，可以继续执行其他的计算逻辑，从而交叉式地重叠了网络传输与计算过程，有利于改进上层系统的 CPU 有效利用率。

RDMA 依赖于网卡的缓存资源提供高性能的数据转发，而 RDMA 网卡上有限的硬件资源带来了扩展性问题。RDMA 采用软硬件倾斜分工的方式实现协议栈卸载，在硬件上通过网卡缓存维护连接 QP 上下文状态、虚拟到物理地址的映射、可靠传输机制等，在软件上仅维护资源的创建。然而，RDMA 网卡缓存资源是有限的，在高并发场景下 RDMA 会因为网卡缓存资源竞争（如并发 QP 争抢缓存、大量 4KB 内存页单元的页表项）而产生大量硬件缓存的未命中（Cache Miss）问题，触发了大量昂贵的 DMA（Direct Memory Access）Read 操作，导致了 RDMA 性能的严重下降。在 Mellanox ConnectX-5 网卡内，每个连接状态需要约 375 B，而网卡共有约 2 MB 的缓存存储连接状态、内存页表映射和其他数据结构，在 5000 个并发连接下会损失约 50% 的吞吐[18]。

2. PCIe NTB

PCIe 非透明桥（Non-Transparent Bridge，NTB）是一种特殊类型的 PCIe 桥设备，可以像“桥”一样将多个分离的计算机系统连接到同一个 PCIe 互连网络上。PCIe NTB 基于硬件上可靠 PCIe 事务层协议的地址转译，将远端主机的内存地址映射到本地 NTB 设备地址空间，通过内存映射 I/O（Memory-Mapped I/O，MMIO）将 NTB 设备内存地址映射到本地主机内存地址空间，从而允许本地应用进程绕过远端主机 CPU 直接访问内存，提供了超低时延、接近 PCIe 带宽的高吞吐。PCIe 互连最初用于将各种 PCIe 设备连接到以 CPU 为中心的计算机系统，将设备内存和主机内存映射到同一个内存地址空间。而 PCIe NTB 使得基于 PCIe 互连的高性能主机间通信成为可能。从硬件可用性的角度，PCIe NTB 既可以内置嵌入到 CPU 处理器中[19, 20]，也可以实现在 PCIe 交换机芯片中[21, 22]，允许机器之间通过可插拔的 NTB 适配器和外部电缆实现 PCIe 互连。从拓扑的角度，一个集成 PCIe NTB 芯片的 PCIe 集群交换机可以将十几甚至几十个机器高速互连起来[23]，适用于扁平化、组网相对简单的机架内通信。

由于不需要 PCIe 协议与网络协议之间的转译，基于 NTB 的轻量级 PCIe 互连相比于 RDMA 是一种理想的高速机架内网络技术，可以提供接近纳秒的超低时延和接近 PCIe 带宽的高吞吐。数据中心内超低时延、高吞吐的机架规模通信已经成为分布式系统提供高质量服务的基础，时延越低越好。主流的协议栈硬件卸载技术如 RDMA 和用户态 I/O 如 DPDK（Data Plane Development Kit），虽然可以通过绕过操作系统内核来降低时延，但仍然依赖于复杂冗余的分层协议栈，引入了不可避免的协议层之间的转译开销，如 PCIe 协议与网络协议之间的转译，对于跨机器之间 PCIe 连接的外围设备的以太网访问，进一步增加了数据路径上协议转译的次数。而 PCIe NTB 是一种基于 PCIe 协议的轻量级互连技术，天然地不存在 PCIe 协议与网络协议之间的转译开销，PCIe 协议自底向上包括 PCIe 物理层、PCIe 数据链路层和 PCIe 事务层，也是一种基于 PCIe 事务层的数据包（Transaction Layer Packet，TLP）的数据通信网络，通过 PCIe 事务层协议和数据链路层协议提供可靠传输和 QoS（Quality of Service，QoS）机制，每个数据包的 PCIe 读写操作均是事务性的。本文第 5 章的实验评估展示了 PCIe NTB 相比于 RDMA 可以实现 2.3~5.6 倍的时延加速。由于 PCIe NTB 的实现默认假设与 PCIe 总线标准对齐，可以提供近乎 PCIe 带宽的吞吐[24]。本文关注于用户空间 NTB 驱动[13] 以摆脱系统内核的复杂性。

由于 PCIe NTB 最初是为跨 PCIe 域的设备通信（如设备共享[23]）而设计的，原生 PCIe NTB 提供了相对低级的网络功能抽象，缺少面向多连接的数据传输模型，很难直接应用于机架规模系统。具体包括：

- **低级编程抽象：** PCIe NTB 提供了面向全局的共享内存和内存语义单边 Write/Read 操作原语，从而支持数据零拷贝、远端 CPU 绕过特性。数据传输的一致性依赖于单边 Write/Read 原语对 TLP 数据包的事务性保证。然而，由于原生 NTB 驱动默认假设独占 NTB 端点（Endpoint）设备来实现主机间设备共享[23]，PCIe NTB 采用了面向单用户进程的编程模型，从而缺少了类似于 TCP 套接字和 RDMA QP 的连接级抽象。在控制面，通过 NTB 端点之间基于寄存器的复杂协商来实现两端映射的全局内存地址元数据交换；在数据面，仅支持本地单应用进程操作 NTB 映射出的远端内存，且不支持类似 POSIX *epoll* 和 RDMA CQ 事件通道的异步通信。

- **有限传输资源：**PCIe NTB 最重要的传输资源是 NTB 映射出的内存空间，但当前 NTB 硬件上内存转译的复杂性限制了 NTB 共享内存空间大小，有限的 NTB 内存资源引入了扩展性问题。如 Intel Xeon 处理器内置 NTB 仅支持 512MB 内存空间[25]。而且，原始 PCIe NTB 驱动存在单进程独占控制的默认假设，使得原生 PCIe NTB 缺少类似于 RDMA MR 抽象的面向多用户进程隔离的内存管理。

1.2.2　分布式系统与协议栈

1.　分布式系统

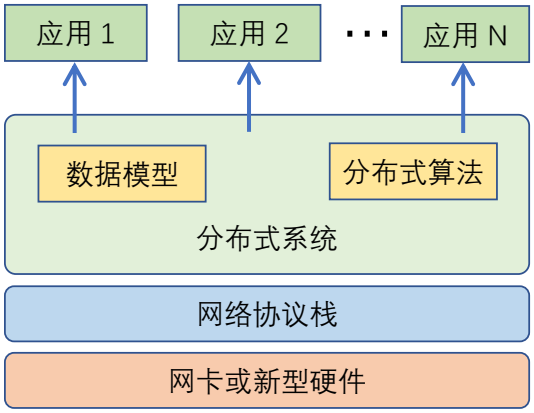


图 1-3　分布式系统概览示意图

分布式系统是数据中心的重要组成部分，对下通过网络池化管理计算和存储资源，对上通过网络连接的多节点协同为上层应用提供数据模型访问和分布式算法等服务。根据对性能和资源开销需求的不同，分布式系统大致可以分为两类：一种面向数据模型提供大规模数据的访存和可用性保证，依赖高带宽、低时延的数据传输，内存开销大，如面向数据模型存储的 NoSQL 系统，称为吞吐/内存开销敏感型；另一种利用算力驱动的分布式算法提供数据的完整性和一致性保证，依赖频繁的点到点通信驱动分布式共识，CPU 开销大，如面向分布式共识服务的区块链系统，称为时延/CPU 开销敏感型。如上文所述，两种类型的系统在海量数据规模和通信密集的背景下对网络的性能提出了更高的要求。

吞吐/内存开销敏感型系统在吞吐性能、可用内存资源、扩展性方面均受制于低效的传统 TCP/IP 网络。在性能方面，系统通常采用客户端-服务端架构，通过网络为上层应用提供基于数据模型的大规模数据增删改查操作，依赖于数据操作的端到端高吞吐来保证服务质量，而传统网络在数据路径上存在多次内存拷贝、系统中断处理等昂贵代价，单核吞吐很难达到线速，严重降低了数据模型访存的端到端吞吐。在内存资源方面，为了提高数据模型基本操作的处理速度，系统服务节点通常把部分或全部数据维护在内存或新型非易失性内存中，而传统网络数据路径的多次内存拷贝，在大规模的数据交互下带来了高昂的内存资源消耗，争抢了数据模型存储所需的可用内存资源。在扩展性方面，系统通常采用基于数据模型的数据分片技术，将数据的多个副本同步到多个分片服务节点，以提高数据的可用性和系统横向扩展能力，多分片节点之间数据副本同步速度被慢速的传统网络约束，不利于数据的一致性和系统扩展性；同时，在高并发请求负载场景下，传统网络由于数据路径上频繁

的中断处理、系统调用、锁竞争等引入了较大的 CPU 开销，从而限制了服务节点的并发处理能力。

时延/CPU 开销敏感型系统在时延性能、可用 CPU 资源、扩展性方面也受到了传统 TCP/IP 网络的严重影响。在性能方面，系统用时延定义服务质量，通常需要在给定的时延约束内对来自上层应用的计算任务或请求作出响应。传统网络由于操作系统内核主导数据路径协议栈的处理，引入了较大的时延开销，从而降低了服务质量。在 CPU 资源方面，系统消耗大量算力资源来驱动分布式算法或协议，如分布式共识协议，依赖于计算节点之间频繁的点到点通信。密集的共识通信使得传统网络的系统调用、内存拷贝、中断处理等 CPU 开销被放大，直接争抢了计算任务所需的可用计算资源，进而影响了任务完成时间。在扩展性方面，系统通常在新计算节点加入时需要通过网络从其他节点同步计算任务状态和数据，其同步效率也受制于传统网络上的时延开销。

2. 协议栈

网络协议栈为上层分布式系统提供丰富的网络功能抽象，是实现高性能数据传输的基础。如前文所述，当前主流的高性能网络协议栈主要分为协议栈硬件卸载（如商用 RDMA）和用户空间 I/O 技术（如 mTCP[11]）。从软硬件分工的角度，RDMA 把整个网络协议栈有状态地卸载固化到网卡硬件上，而 mTCP 则把整个 TCP/IP 协议栈实现在用户态软件层，从而绕过操作系统内核以提供低时延和高吞吐。

然而，这些高性能协议栈并非适用于所有场景，在拓扑扁平化、时延非常敏感的机架规模网络场景下仍然存在性能局限性。一方面，数据中心内超低时延、高吞吐的机架规模网络已经成为提供高质量系统服务的基础，且通信时延越低越好。另一方面，主流高性能协议栈仍然依赖于冗余且厚重的分层协议栈，尤其需要 PCIe 协议与网络协议之间的转译，从而导致了不同协议层之间的转译延迟开销。例如，RoCEv2 将 RDMA IBTA 协议构建在基于以太网 UDP 之上，在 PCIe 总线协议与 IBTA 协议之间、IBTA 协议与 UDP 之间均存在协议转译，在互连主机之间的一次单向数据移动需要经历至少 4 次协议转译，导致了不可忽视的（微秒级）时延开销。同时，在一个机架内，跨主机对 PCIe 连接的外围设备的以太网访问进一步增加了数据路径上协议转译的次数。因此，面向机架规模网络消除或摆脱 PCIe 协议与网络协议之间的转译，从而为机架规模系统提供更低的通信时延，是一项关键技术挑战。

1.2.3　面向内核旁路网络的系统与协议栈关键技术与问题

从内核旁路网络增强系统的角度，由于商用 RDMA 网络与传统 TCP/IP 网络的特性存在很大的区别，因而基于商用 RDMA 网络的分布式系统构建与基于传统内核态 TCP/IP 栈的分布式系统构建有比较大的不同。传统分布式系统通常针对传统 TCP/IP 网络进行了过多的设计，尚未充分考虑商用 RDMA 网络特性引入的丰富设计空间。简单朴素地把传统分布式系统移植运行在商用 RDMA 网络上，既难以充分发挥商用 RDMA 网络的巨大性能优势，也无法有效地规避或缓解商用 RDMA 的不足。图 1-4 列出了 RDMA 特性对分布式系统构建技术的关键影响。在分布式系统的性能优化技术方面，首先面向吞吐/内存开销敏感型系统，RDMA 内存语义特性可以利用 RDMA 内存区域（MR）组织数据模型，并扩展基于内存语义的数据基本操作，RDMA RC 模式下的单核线速转发能力有利于提高数据操作端到端

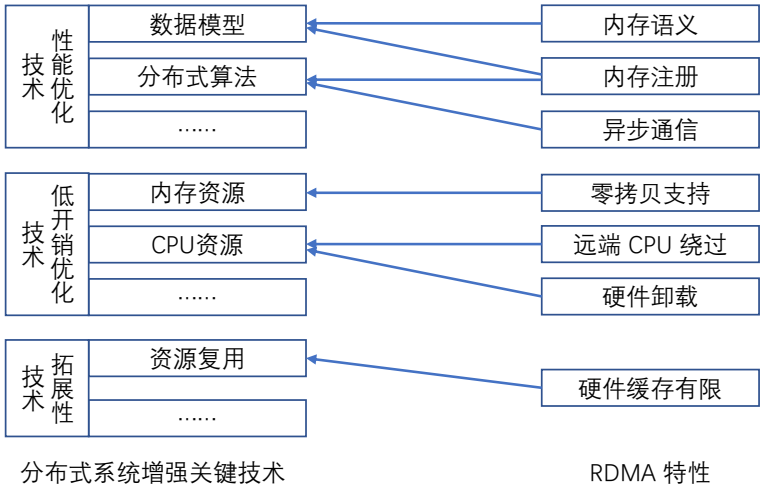


图 1-4　RDMA 特性对分布式系统增强技术的主要影响

吞吐。其次面向时延/CPU 开销敏感型系统，RDMA 内存语义单边原语特性可以提供低时延的请求响应时间，RDMA 异步通信特性可以交叉重叠分布式计算和数据同步传输，有助于提升 CPU 资源的有效利用率。除此之外，RDMA 内存注册的时延开销大，这要求分布式系统在数据操作路径上要避免频繁的 RDMA 内存注册/反注册操作。在分布式系统的低开销技术方面，首先面向吞吐/内存开销敏感型系统，RDMA 零拷贝特性避免了端到端数据路径上的内存拷贝开销，有利于提高存储数据模型所需的可用内存资源。其次面向时延/CPU 开销敏感型系统，RDMA 单边原语具有绕过远端 CPU 的特性，硬件卸载特性避免了软件层网络栈处理的 CPU 开销，均有助于增加分布式算法所需的可用算力资源。在分布式系统的扩展性技术方面，RDMA 网卡缓存资源的有限性要求分布式系统在系统设计时考虑新的扩展性维度，即资源复用维度；同时，RDMA 的 CPU 绕过特性也有利于提高服务端单核并发处理能力。具体而言，RDMA 是基于有状态的协议栈卸载技术，依赖于网卡内硬件缓存资源来维护连接上下文状态、内存映射页表、可靠传输上下文状态等。在网卡缓存资源消耗殆尽时，会产生大量缓存未命中，此时网卡需要通过 PCIe 总线对主机内存进行大量 DMA Read 操作，以同步主机内存中的状态和页表数据到网卡缓存中，直接影响了高并发场景下的通信性能，不利于系统扩展性。而传统分布式系统并未充分考虑 RDMA 网卡缓存对系统通信扩展性的影响。因而，面向商用 RDMA 网络的系统增强在横向、纵向扩展性技术中均需要考虑资源复用维度。

从基于内核旁路网络新型硬件的协议栈角度，由于商用 RDMA 并非适用于所有场景，在拓扑简单、时延越低越好的机架规模通信场景下[26]，PCIe 协议与网络协议的转译引入了微秒级的时延开销，而 PCIe NTB 互连则消除了这一协议转译开销，是一种适用于机架规模通信的理想高速网络技术。图 1-5 列出了 PCIe NTB 特性对轻量级协议栈设计关键技术的的主要影响。在协议栈的性能优化技术方面，PCIe NTB 具有不需要 PCIe 协议与网络协议之间的转译和绕过操作系统内核的特性，有利于协议栈提供接近纳秒级的超低时延，但由于原生 PCIe NTB 驱动的单进程独占假设，不支持面向多租户/系统混部的性能隔离。在协议栈的低开销技术方面，PCIe NTB 的零拷贝和 CPU 绕过同样有利于协议栈改善资源效率。在协议栈的兼容扩展性技术方面，如前文所述，由于 PCIe NTB 最初是用于 PCIe 域间的设备通信和共享[24]，原生 PCIe NTB 的单进程编程模型特性缺少类似 TCP 套接字和 RDMA QP

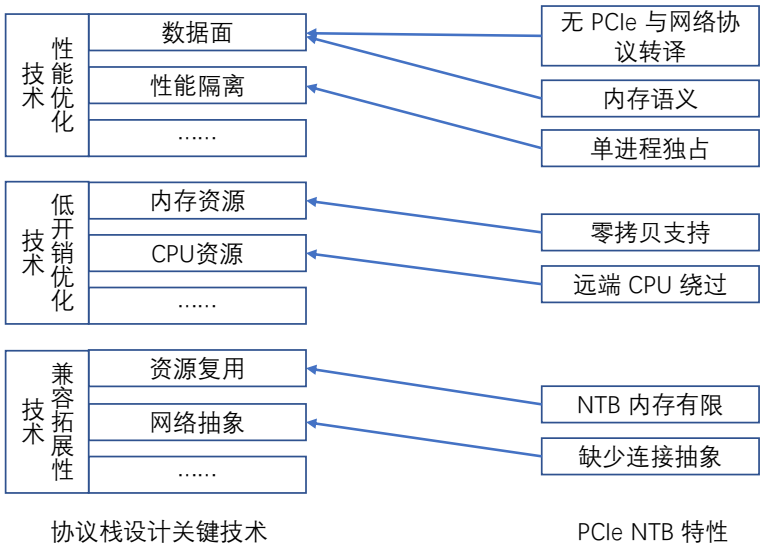


图 1-5　PCIe NTB 特性对协议栈设计的主要影响

的连接级网络抽象，不利于协议栈面向多用户进程提供兼容的网络编程抽象；同时，有限的 NTB 映射内存作为关键的网络传输资源，限制了数据面扩展性，要求协议栈设计从软硬件协同的角度尽可能复用 NTB 内存资源。总而言之，原生 PCIe NTB 缺少通用的网络功能抽象，如何基于 PCIe NTB 硬件设计轻量级的协议栈，为机架规模系统提供兼容、扩展和多租户性能隔离的网络功能，是一个重要问题。

基于内核旁路网络的分布式系统增强和协议栈设计的主要问题可以概括如下：

- **性能优化不匹配：**在系统增强方面，传统分布式系统多针对慢速 TCP/IP 网络的特性进行设计优化，如 TCP 段卸载[16] 和发送窗口缩放来提升吞吐等，不再适用于内核旁路网络。而 RDMA 内存语义零拷贝特性依赖于单边 RDMA 通信原语，并基于 RDMA 内存区域重新组织吞吐/内存敏感型系统的数据模型，才能为其带来兼顾高吞吐和低内存开销的收益；RDMA CPU 绕过特性同样依赖于单边 RDMA 通信原语，并结合 RDMA 异步通信机制重新设计驱动分布式算法的点到点通信，才能为时延/CPU 敏感型系统带来兼顾低时延和低 CPU 开销的收益。因而，吞吐/内存敏感型、时延/CPU 敏感型系统分别需要在数据模型组织、分布式算法同步上考虑与 RDMA 内存语义单边原语相适应的网络/系统的协同设计。

- **有限资源难扩展：**传统分布式系统主要针对内核态 TCP/IP 网络设计，而 TCP/IP 网络的协议栈状态管理维护在内核空间，不存在因硬件缓存资源竞争或有限的硬件映射内存竞争导致的性能扩展问题。然而，与传统网络不同，RDMA/PCIe NTB 均通过协议栈硬件卸载实现高性能通信，RDMA 依赖于网卡缓存来维护跟踪连接状态、页表映射等信息，会因高并发连接处理和大量的 RDMA 内存注册而造成网卡缓存不命中、网卡与主机内存之间频繁的 DMA Read 操作，最终导致吞吐大幅下降；PCIe NTB 映射的内存空间有限，也会因高并发处理下的 NTB 内存资源争抢导致数据面性能下降。因而，在基于内核旁路网络的系统增强与协议栈设计中，资源的局限性对数据面性能扩展的影响不可忽略。

- **通信抽象难兼容：**传统 TCP/IP 网络和商用 RDMA 均提供面向连接的网络功能抽象，

如网络套接字和 RDMA QP。然而，原生 PCIe NTB 由于固有的 NTB 设备独占式假设，采用单用户进程编程模型，尚未提供连接级网络抽象，这为软件层管理连接上下文状态提供了可能。因而，面向 PCIe NTB 的轻量级协议栈需要采用软硬件协同的方式屏蔽低级编程抽象，提供兼容的面向多用户的网络抽象。

- **混部性能难隔离：**机架规模网络需要应对不同流量模式的系统混部场景，在共享 PCIe NTB 传输资源的情况下引入了性能隔离挑战。然而，原生的 PCIe NTB 架构缺少面向多应用/租户的性能隔离机制，NTB 资源的共享不可避免地引入了线头阻塞和多个复用单元之间负载不均衡的问题。在基于 PCIe NTB 的轻量级协议中，设计基于有限 NTB 内存的高效资源共享机制以支持性能隔离是有挑战性的。

综上所述，如何充分利用内核旁路网络的特性和通信原语进行系统增强和协议栈设计，是为数据中心规模的分布式系统提供高性能、资源高效、兼容扩展的高质量服务的关键问题和挑战。因此，本文拟针对面向内核旁路网络的系统增强和轻量级协议栈设计关键技术进行深入研究。在系统增强方面，本文聚焦在利用商用 RDMA 网络增强吞吐/内存敏感型系统、时延/CPU 敏感型系统，分别以基于文档数据模型的 NoSQL 系统、基于分布式共识协议的区块链系统作为典型案例展开研究。在轻量级协议栈设计方面，本文聚焦在利用 PCIe NTB 硬件设计与实现面向机架规模通信的高性能、兼容扩展、性能隔离的网络功能，以消除 PCIe 协议与网络协议的转译开销，避免商用 RDMA 在机架规模通信的性能局限性。

1.3　本文研究内容与核心贡献

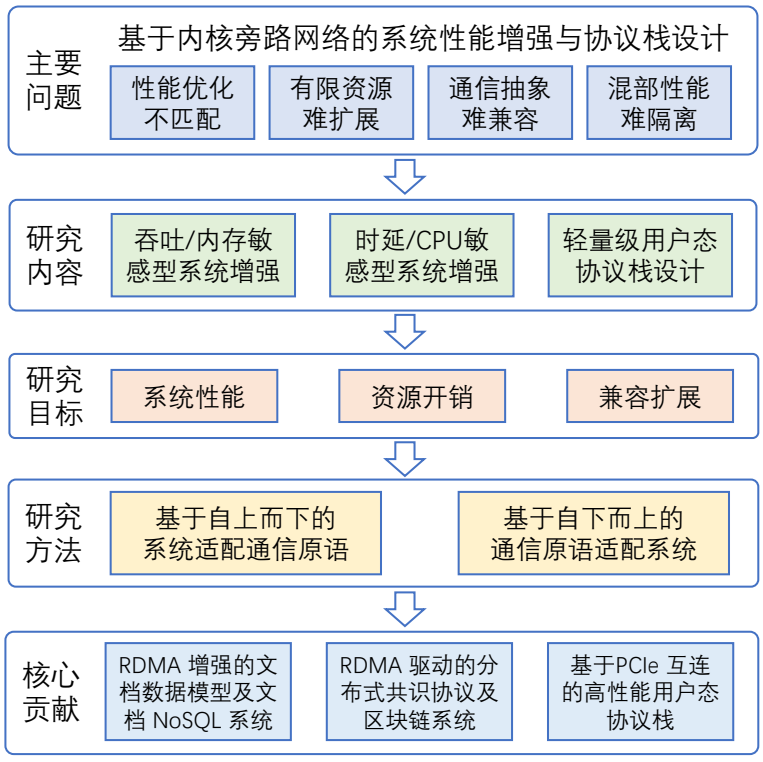


图 1-6　本文的主要问题，研究内容，研究目标，研究方法和核心贡献

1.3.1　研究内容

为了解决上述关键问题，本文旨在系统性地研究基于内核旁路网络的系统性能增强和高性能协议栈设计，提出协同优化性能、资源开销、兼容扩展性的核心研究目标。围绕该目标，本文从网络与系统协同、软硬件协同两个角度重新思考分布式系统与内核旁路网络的结合方式，提出基于自上而下的系统适配通信原语、基于自下而上的通信原语适配系统两种研究方法，在内核旁路网络加速吞吐/内存开销敏感的系统、加速时延/CPU 开销敏感的系统以及轻量级用户态协议栈设计三个方面分别展开研究。本文的主要研究内容包括：

1. 在优化吞吐/内存开销敏感的系统方面，研究适应于 RDMA 内存语义单边原语特性的 NoSQL 文档数据模型。与传统面向流的 TCP 网络套接字不同，内核旁路的 RDMA 单边操作原语是面向内存语义的，允许网卡绕过远端 CPU 直接操作远端主机内存，该特性可用于文档数据模型设计，从而可以将基于内存语义的增删改查数据操作间接地卸载到 RDMA 网卡硬件。同时，RDMA 内存的注册与反注册操作开销很大，引入了不可忽视的时延，频繁的注册/反注册操作会降低内存效率，阻碍了分布式系统对 RDMA 内存语义单边原语特性的利用。因而，本文从网络与吞吐/内存开销敏感系统协同设计的角度，研究高性能、内存高效且高可用的 NoSQL 文档数据模型。本文首先研究利用内存语义的 RDMA 单边写入（Write）原语扩展文档数据模型的操作语义，在提供高性能的同时，改进文档 NoSQL 服务节点在高并发请求下性能的扩展性；然后，研究结合 RDMA 内存注册/反注册的特征研究负载感知的 RDMA 缓冲区注册机制，以降低内存开销；最后，研究传输链路的容错机制和基于 RDMA 单边原语的多分片节点之间的操作日志同步协议，以改进链路和数据的可用性。

2. 在优化时延/CPU 开销敏感的系统方面，研究适应于 RDMA 的 CPU 旁路和异步通信特性的分布式共识协议。一方面，分布式共识协议既依赖于频繁的点到点通信进行共识确认，又依赖于大量计算资源进行共识哈希计算，而传统内核态 TCP 网络栈在数据路径上引入较大的 CPU 开销，会影响共识效率，带来了较大的通信时延和 CPU 开销。另一方面，单边 RDMA 原语提供低时延且绕过远端 CPU，RDMA 异步通信特性可以交错重叠通信与计算，同时 RDMA 硬件网卡的缓存资源是有限的。因而，本文从网络与时延/CPU 开销敏感系统协同设计的角度，结合 RDMA 单边原语和异步通信特性研究驱动分布式共识的点到点通信机制；同时，针对 RDMA 硬件缓存资源的局限，研究基于 RDMA 共享接收队列和混合通信原语的新节点引导协议设计技术。

3. 在设计系统依赖的网络协议栈方面，研究基于 PCIe 互连的轻量级用户态协议栈设计。不同于依赖厚重分层协议栈的商用 RDMA 网络，PCIe 非透明桥（NTB）不需要 PCIe 协议与网络协议之间的转译，可以互连 PCIe 连接的机器或 PCIe 设备，是适用于机架内通信的理想高速网络技术，实现了比 RDMA （2.3 至 5.6 倍）更低的通信时延。但是 PCIe NTB 缺少通用的网络功能支持。因此，本文从软硬件协同的角度，首先研究面向控制平面与数据平面分离的用户级间接层架构；然后研究适应于原生 PCIe NTB 原语的高性能且兼容的通信抽象设计；接着结合并行化思想研究适应于 PCIe NTB 资源能力的可扩展数据平面设计；最后研究适应于 PCIe NTB 资源共享的多租户性能隔

离机制，并研究了基于 PCIe NTB 的机架内网络与基于 RDMA 的机架间网络混合部署的潜在优势。

1.3.2　核心贡献

对应于上述研究内容，针对内核旁路网络与传统分布式系统通信层不匹配的问题，本文系统性地分析了内核旁路网络的特性和分布式系统的设计目标之间的关系，从自上而下的系统匹配通信原语、自下而上的通信原语匹配系统两个层面出发，提出了面向内核旁路网络增强系统和高性能协议栈的三项关键创新设计技术，协同优化了分布式系统的性能、资源开销和兼容扩展性，促进了内核旁路网络的在分布式系统上的性能优势发挥和广泛应用。本文的核心贡献包括：

1. **从网络和吞吐/内存开销敏感系统协同设计的角度，提出了一种 RDMA 增强的高性能、内存高效、高可用的文档数据模型，并实现了对应的文档 NoSQL 系统 RMongo**。为了传输链路的容错性，本文设计了 RDMA 上下文检测算法，通过检测 RDMA 链路的可用性，从而决定两个节点之间使用 RDMA 或传统 TCP/IP 通信；设计了负载感知的 RDMA 缓冲区注册机制，以实现内存高效的 RDMA 通道内存区域管理；RMongo 基于内存语义的单边 RDMA 通信原语，来扩展面向文档数据模型的增删改查操作语义，提升了文档 NoSQL 的端到端性能；为了文档数据模型的高可用，我们结合单边 RDMA 原语和异步完成事件通道，重新设计了多分片节点之间的操作日志同步协议。实验表明，相比传统的基于 Linux TCP 的文档 NoSQL，RMongo 可提升端到端数据操作的吞吐量达 30%。

2. **从网络和时延/CPU 开销敏感系统协同设计的角度，提出了一种 RDMA 驱动的高性能、CPU 高效、可扩展的分布式共识协议，并在此基础上实现了对应的许可区块链系统 BoR**。本文利用单边 RDMA 内存操作原语和基于 RDMA 完成事件通道的异步事件模型，重新设计驱动分布式共识的点到点通信，绕过服务端 CPU，使 BoR 既实现了更低的共识同步时延，又最小化了通信带来的 CPU 开销；利用混合 RDMA 原语和共享接收队列设计了新节点引导协议，BoR 降低了新节点初始化同步历史区块数据的时延，提高了共识网络的扩展性。实验表明，在不同规模的工作负载下，相比于基于 Linux TCP 的区块链系统，BoR 的新节点共识过程降低了 CPU 开销达 26.4%，降低了区块同步时延达 20.2%。

3. **从软硬件协同设计的角度，提出了一种基于 PCIe 互连的、控制平面与数据平面分离的轻量级用户态协议栈 NTSocks**。设计了一个高性能、软硬件协同的用户级间接层，在屏蔽了底层高速网络 PCIe 互连和 RDMA 异构性的同时实现了兼容、多核扩展、性能隔离的网络功能，有效隔离不同流量模式的系统性能，并实现了对应的通信框架。考虑到 PCIe 非透明桥（Non-Transparent Bridge，NTB）互连比 RDMA 少了 PCIe 协议与网络协议之间的转译开销，实现了 2.3~5.6 倍更低的网络时延，且组网拓扑更简单，本文设计了面向机架内通信的、基于 PCIe NTB 互连的用户级轻量级协议栈，实现了比商用 RDMA 网络更低的机架内通信时延，且在一定程度上缓解了 RDMA 机顶交换机出端口 Incast 拥塞。通过设计兼容的类套接字抽象，NTSocks 以用户态运行

时库的形式解决了原生 PCIe NTB 抽象不匹配的问题。为了数据平面的多核扩展性，NTSocks 基于并行化思想实现了 CPU 核心驱动的数据平面模型。为了公平高效的资源共享，NTSocks 设计实现了面向多租户的性能隔离机制。尽管 PCIe NTB 最初是为跨 PCIe 域的设备通信而设计，NTSocks 展示了一个灵活的用户级间接层，可以在提供通用网络功能的同时实现接近裸机 NTB 的网络性能。实验表明，相比 Linux TCP 和 RDMA 套接字，NTSocks 降低微基准测试时延分别到 1/32 和 1/4.4，降低键值存储系统的端到端时延分别到 1/24.5 和 1/1.58，降低了 Nginx HTTP 文件服务响应时延到 1/6.7。

1.4　论文结构安排

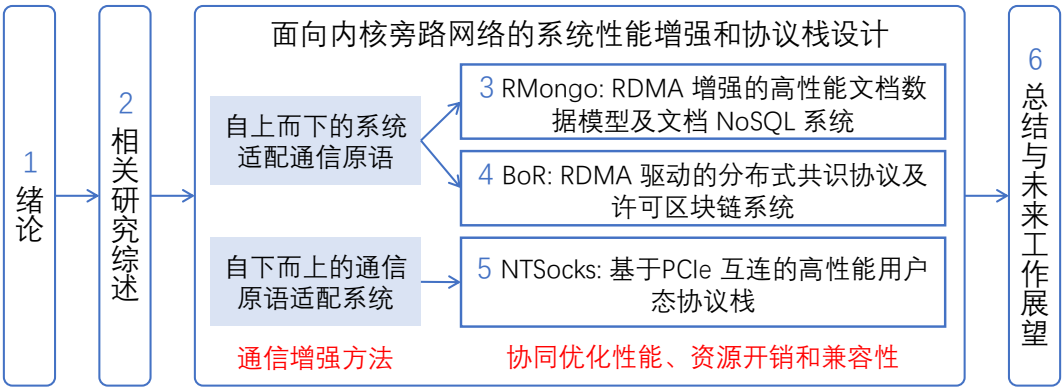


图 1-7　本文内容组织结构

全文围绕基于内核旁路网络的系统增强和协议栈设计的选题可分为六个章节，各章节的主要内容如下：

第 1 章为本文绪论，首先介绍了基于内核旁路网络的系统性能增强和协议栈设计选题的研究背景和意义，及面向内核旁路网络的系统与协议栈设计所面临的关键问题与挑战，然后介绍了围绕内核旁路网络增强系统和设计协议栈的研究内容与核心贡献，最后介绍了全文章节的组织结构。

第 2 章为本文的相关研究综述，分别介绍了基于内核旁路网络的分布式系统加速和通信协议栈设计的相关研究工作。

第 3 章从网络和吞吐/内存开销敏感系统协同设计的角度，提出了一种 RDMA 增强的高性能、内存高效、高可用的文档数据模型，以及对应的文档 NoSQL 系统实现 RMongo。

第 4 章从网络和时延/CPU 开销敏感系统协同设计的角度，提出了一种基于商用 RDMA 网络的高性能、CPU 高效、可扩展的分布式共识协议，以及对应的许可区块链存储系统实现 BoR。

第 5 章提出基于 PCIe NTB 互连的高性能用户态协议栈，介绍了一个控制平面与数据平面分离、软硬件协同的高性能用户级间接层，以及对应的通信框架实现 NTSocks。

第 6 章针对全篇的研究内容和创新性贡献进行了总结，并讨论了面向内核旁路网络和系统增强的未来潜在研究要点和研究价值。

第 2 章　相关研究综述

本章介绍了利用内核旁路网络在分布式系统加速和高性能协议栈设计方面的已有研究工作，包括基于硬件卸载和用户态 I/O 的内核旁路网络相关研究。首先介绍了基于 RDMA 加速的吞吐/内存敏感型系统的背景工作，聚焦在吞吐/内存敏感的 NoSQL 系统。然后介绍了基于 RDMA 加速的时延/CPU 敏感型系统的背景知识和研究工作，聚焦在面向时延/CPU 敏感的分布式共识协议和区块链系统。最后介绍了机架规模网络通信与用户态协议栈的背景和研究工作。

2.1　基于 RDMA 加速的吞吐/内存敏感型系统

本节对新型网络硬件、内核旁路网络、RDMA 协议栈优化和 RDMA 驱动的 NoSQL 系统等背景知识和相关工作进行了广泛的调研与讨论。

新型网络硬件：RDMA 网络分为三种类型：Infiniband[27]、RoCE 和 iWARP[28]。Infiniband 是为 RDMA 设计的网络，从硬件层面保证可靠传输。RoCE 和 iWARP 是构建在以太网上的 RDMA 协议的不同硬件实现，对上层应用保留了相同的网络编程抽象[29]。特别地，RoCE 协议有 RoCEv1 和 RoCEv2[30] 两个版本。两个版本的主要区别在于，RoCEv1 是基于以太网链路层实现的 RDMA 协议，而 RoCEv2 基于以太网 TCP/IP 协议栈中的 UDP 协议层实现。文献[10, 31] 中的调查表明，与 RoCE 和 iWARP 相比，Infiniband 具有更好的性能。但是，Infiniband 的网卡和交换机更昂贵，且主要面向高性能超算网络。相比之下，RoCE 和 iWARP 的硬件实现与以太网兼容，只需要专用网卡和支持优先级流量控制的以太网交换机，部署成本部相对更低。尽管具有类似的规范，iWARP 比 RoCE 更昂贵、更复杂。此外，iWARP 的吞吐量比 RoCE 低 4 倍，时延高 3 倍[31]。因此，由于物理环境的限制，本研究采用 RoCEv2 网络。

内核旁路网络：内核旁路网络通过将数据包处理移至用户空间或将网络协议栈卸载到专用 NIC （Network Interface Card）中，消除了操作系统网络协议栈的开销以及应用程序缓冲区和操作系统内存缓冲区之间的数据拷贝。内核旁路网络的主流技术包括 DPDK[32] 和 RDMA。TCP/IP 网络协议栈在 DPDK 中向上移动到用户空间。DPDK 利用用户空间 I/O（Userspace IO，UIO）将 NIC 设备内存映射到用户空间。DPDK 利用大页内存显著地降低了高并发内存访问负载下 TLB 缓存未命中的次数。同时，DPDK 规避了系统中断处理，而是使用基于 CPU 核的轮询模式来直接访存硬件网卡中的数据包。但是，DPDK 中的用户空间网络协议栈仍然涉及 CPU 干预，DPDK 的并发能力在很大程度上取决于 CPU 内核的数量，在流量负载较低的情况下会导致不必要的 CPU 核资源的开销。对于商用 RDMA，网络协议栈被直接卸载到专用网卡硬件中，而无需任何 CPU 参与。RDMA 的高带宽和零拷贝特性为吞吐/内存敏感系统提供了高吞吐、超低时延的性能优势，RDMA 网络收发报文的速率完全

由 RNIC（RDMA NIC）的转发能力决定。所以 RMongo 的原型系统是构建在内核旁路的高性能商用 RDMA 网络，充分将吞吐/时延敏感的文档数据模型与 RDMA 传输特性进行协同设计。

RDMA 协议栈优化：最近的一些研究工作[31, 33, 34]集中在 RDMA 上的协议优化。与之前 RDMA 的端到端单路径传输不同，MP-RDMA[34] 提出了多路径 RDMA 传输，它充分利用了数据中心丰富的传输路径。IRN[31] 提出了一种更有效的基于 Bitmap 数据结构的选择性丢包恢复和端到端流量控制机制，以传输中的数据包数量为界，这消除了对基于优先级流控（Priority-based Flow Control，PFC）的无损网络的依赖，并通过仿真实验证明了 IRN 的网络性能优于 RoCE 网络。研究工作[33] 展示了不可靠发送/接收和 RDMA 写入无连接数据报的统一设计，它比面向连接的 RDMA 传输实现了更高的性能和可扩展性。以上研究主要涉及 RDMA 的底层机制，而缺乏对基于 RDMA 的事务性资源管理的关注。RMongo 基于商用 RoCEv2 网络，提出了一种新颖的负载感知缓冲区注册算法，用于高效的内存管理。

RDMA 驱动的 NoSQL 数据库：由于对海量半结构化/非结构化数据的存储和分析的强烈需求，NoSQL 数据库有望提供更高的吞吐量、更低的时延和 CPU 开销。RDMA 驱动的 NoSQL 系统，尤其是键值存储[35–44] 和图存储[45]，在数据中心变得越来越流行。

Pilaf[42] 利用单边 RDMA Read 原语来优化 Get 请求，最大化的降低了键值服务端的 CPU 开销，并提出自验证数据结构以消除客户端和服务器之间的读写竞争。在 FaRM[44] 中，集群中的内存通过 RDMA 的内存语义特性，被透明地暴露为全局共享的内存地址空间。FaRM 支持 RDMA 上的无锁读取、函数传送和对象搭配，以实现更高效的键值存储或其他应用程序的事务。HydraDB[39] 展示了一种新型的基于 RDMA 的通用键值中间件，并从多个方面提升了高性能内存键值存储，包括基于 RDMA Write 原语的消息传递、基于 RDMA Read 的 Get 操作、远程指针共享和轻量级的一致性机制。此外，HydraDB 有效利用多核系统来充分发挥 RDMA 的潜力。与 FaRM 和 Pilaf 等基于 RDMA Read 原语的键值系统不同，HERD[38] 混合使用单边 RDMA 写入和双边发送/接收原语来执行 GET 和 PUT 操作，这减少了网络往返并实现比之前的 FaRM 和 Pilaf 高 2 倍的吞吐量。在 Wukong[45] 中，提出了 RDMA 驱动的分布式 RDF（Resource Description Framework）图存储系统，以实现对大型 RDF 数据集的 RDF 查询的更低时延、更高吞吐量和并发性。具体来说，RDMA Read 用于小规模的 RDF 查询，而 RDMA Write 用于大规模的非选择性查询。然而，据我们所知，仍然没有尝试使用 RDMA 加速基于文档的 NoSQL 系统。因此，我们专注于利用内存语义的单边 RDMA 原语来增强吞吐/内存敏感的文档 NoSQL 数据库，设计目标是高吞吐量、低时延和低 CPU 开销。

近些年内核旁路网络是一个新兴趋势，以适应数据密集型计算的需求。我们围绕面向商用 RDMA 网络的吞吐/内存敏感的文档数据模型及对应 NoSQL 系统，系统性地探索研究了 RDMA 网络与吞吐/内存敏感系统的协同设计和性能增强技术。

2.2　基于 RDMA 加速的时延/CPU 敏感型系统

本节介绍了 RDMA 增强的时延/CPU 敏感型系统相关的背景知识和研究工作，重点聚焦在典型的时延/CPU 敏感的分布式共识协议及区块链系统。

时延/CPU 敏感的系统增强。分布式共识协议和区块链系统是典型的时延/CPU 敏感场景。近年来，许多研究工作集中在如何优化分布式共识驱动的区块链的系统性能。在工作[46]

中，作者首先重构了 Hyperledger 和以太坊的架构，然后评估了这些架构的可修改性、安全性和性能，作者认为这三个指标是区块链评估中最重要的指标。在文献[46] 中，作者首先证明了分布式共识协议通过一个优先级约束的对抗延迟，在异步通信过程中满足数据的强一致性和活动性，作者进一步提出了一个抽象的区块链协议来确保安全一致性校验。许多其他研究工作[47–49] 关注于面向分布式共识和区块链系统的性能分析并取得了很大的成就。然而，这些工作主要聚焦在区块链的算法与数据结构的优化，忽视了网络性能对于分布式共识协议和区块链的重要影响。因此，我们以分布式共识协议及区块链作为典型的时延/CPU 敏感型系统，系统性地研究了内核旁路网络与时延/CPU 敏感系统协同设计技术。

一致性的分布式共识协议。由于时延/CPU 敏感的分布式共识算法是区块链技术最重要的基础组件，很多研究工作[50–52] 均聚焦在分布式共识协议的优化。Bitcoin-NG[50] 提出了一种面向下一代 Bitcoin 的易扩展的、拜占庭容错的新区块链协议，使得吞吐仅受到每个节点的网络硬件带宽限制，而时延仅受网络广播时间的限制。在文献[51] 中，作者提出了一个新的框架来监控基于工作量证明（Proof-of-Work，PoW）的区块链，并介绍了共识和网络的各种参数，在设计区块链框架时考虑了现实世界的约束（例如网络传播、不同的块大小、块生成间隔和信息传播机制）。在文献[52] 中，作者提出了一个名为 PoT 的信任证明区块链来减少 PoW 的能量消耗。使用 PoT，只要证明对等节点更可信，它就可以做更少的工作。PoT 是一种典型的基于信任图的共识协议优化解决方案。通过以上工作，之前低效的共识机制得到了很大的改善。然而，已有的研究工作很少关注在面向时延/CPU 敏感的分布式共识协议的网络性能增强。这是因为传统的低效共识协议（如 PoW）主要由共识所需的计算开销主导，而随着分布式共识算法的快速演进，共识算法如权益委托证明（Delegated Proof of Stake，DPoS）的计算速度更快，区块事务的共识吞吐更高，分布式共识协议的网络传输逐渐成为潜在的性能瓶颈。因此，BoR 中采用了绕过内核的 RDMA 网络来弥合性能差距，系统性地探索了面向商用 RDMA 网络的时延/CPU 敏感系统的性能增强技术，设计实现了 RDMA 驱动的低时延、CPU 资源高效的分布式共识协议。

面向商用 RDMA 的系统增强。由于具有高吞吐量、低时延和低 CPU 开销，支持 RDMA 的网络最近被广泛部署在数据中心。在文献[53, 54] 中，提出了一种 RDMA 驱动的 B 树存储架构，以减少网络开销并提高性能。Wukong[45] 提供了一种使用 RDMA 通信进行图探索的快速并发解决方案。它基于不同规模的 RDF 查询使用不同的 RDMA 原语（即 RDMA Read/Write）。在文献[39, 40, 43] 中，HydraDB、Nessie 和 InnerCache 都提出了一种基于 RDMA 的高效键值系统。HydraDB 利用面向 RDMA 传输的高可靠方式为分布式键值系统实现了高吞吐、低时延的端到端键值访存性能。Nessie 是一个支持 RDMA 的键值系统，它在服务请求时提供无服务器解决方案，并将索引和数据解耦。InnerCache 是另一种基于 RDMA 的内存键值存储，它提供了一种巧妙的缓存机制，利用双边 RDMA 通信原语来增强系统性能。其他键值存储系统的相关研究[35, 37, 55] 也取得了不错的效果。此外，RDMA 驱动的系统优化也是一个研究热点[56–58]。但是到目前为止，RDMA 驱动的分布式共识协议的相关研究工作仍然欠缺。因此，BoR 创新性地探索了基于商用 RDMA 网络来增强时延/CPU 敏感的分布式共识协议。

2.3　机架规模网络通信与用户态协议栈

机架级网络通信：由于在线服务需要严格的服务级目标（Service-Level Objective, SLO），学术界和工业界对数据中心网络背后的机架级通信进行了大量研究[5, 59–66]。这些研究通过设计网络路由和拥塞控制算法[59, 64] 或可编程交换机[5, 63] 来实现微秒级时延和百万级吞吐量。NTSocks 与这些工作正交，因为它创造性地利用 PCIe NTB 高速互连通路，为机架规模系统提供易于使用、快速、性能缩放和隔离的轻量级协议栈。相比协议栈硬件卸载的研究工作，NTSocks 通过提供轻量级协议栈，减少协议层之间的转译，进一步降低了网络传输时延，提升了上层机架规模系统的服务质量。

性能隔离和服务质量：面向多租户的数据中心系统通常需要考虑不同系统之间的性能隔离和 QoS 问题。为了解决这些问题，之前的工作[67–71] 提出了多种隔离和 QoS 机制设计，包括集中式的全局服务器管理和调度策略。NTSocks 与 PCIe NTB 的特性紧密结合，基于数据平面并行化的思想，提出了分区抽象作为数据平面多个连接的复用单元，并进一步提出了分区内、分区间层次化协同设计来实现多租户性能隔离。NTSocks 还可以通过控制应用程序和数据面之间的连接级共享内存队列，从而实现多种服务质量 QoS 的策略。

用户态网络栈：大量高性能协议栈设计的相关工作[11, 71–76] 利用高性能用户空间 IO 技术，将厚重的网络协议栈上移到用户空间实现，从而允许用户态进程直接读写网卡设备的方式绕过了复杂的操作系统内核，降低了数据路径上的线程上下文切换开销、锁的竞争开销、高昂的系统调用开销和数据拷贝次数。用户态网络栈的架构形态主要分为库操作系统（即 *library OS*）[11, 72–74] 和集中式微内核架构[71, 75, 76]。前者主要是把完整的网络协议栈功能实现在了一个运行时库中，充分利用每个应用进程的计算进程，但这种方式带来了升级部署复杂、无法从全局视角实现资源共享和性能隔离等问题。而后者主要思想是利用一个用户态的独立进程承载所有的网络协议栈功能，为该进程绑定专用 CPU 核心以直接轮询读写网卡上的数据包，并通过共享内存通道向上层应用进程提供网络接口。这种方式易于透明化部署升级，并且可以更细粒度地实现全局网络资源共享调度和性能隔离。

就像基于微内核架构的 Shenango[76] 侧重于微秒级的 CPU 核调度，Snap[75] 采用微内核架构而侧重于终端主机网络优化，NTSocks 虽然借鉴了集中式微内核架构，但本文的核心创新在于首次将 PCIe 互连作为服务于机架内通信的高速网络技术。总体而言，NTSocks 与微内核架构代表性工作 Google Snap[75] 不同之处在于：

（1） 其主要创新性是把 PCIe NTB 互连作为一种高速网络技术来服务于机架内的网络，而传统的 PCIe NTB 互连主要是用于跨主机之间的设备通信（如设备共享）。

（2） Snap 的一个默认假设是底层网络基于以太网设备，而原生 PCIe 互连构建在 PCIe 事务层协议上，并不具备类似以太网的网络功能如基于 IP 和端口的寻址和路由，所面临挑战是 Snap 很难解决的。因此 Snap 很难直接支持 PCIe 互连，而为 PCIe 互连提供类似以太网的网络功能所面临的挑战正是 NTSocks 要解决的核心问题，也是与 Snap 的根本不同点。

（3） Snap 受限于网络，需要重新实现所有的网络设备驱动，而 NTSocks 不需要对 PCIe NTB 设备驱动的任何改造。

（4） Snap 的网络栈依然存在 PCIe 协议与网络协议之间的协议转译开销，而 NTSocks 避免了这种开销，从而实现比 Snap 更低的（尾）时延。

需要注意的是，由于 Google Snap 系统尚未开源，本文无法通过实验评估来展开 NTSocks 与 Snap 之间的性能分析。但根据 Snap 工作[75] 中的实验评估，Snap 的双边通信操作的微基准时延最低是 10 微秒，而 NTSocks 的微基准时延最低是 2.6 微秒，大约是 Snap 通信时延的 1/4。

第 3 章　RMongo: 基于 RDMA 的高性能文档数据模型及 NoSQL 系统

本章的主题是基于 RDMA 增强吞吐/内存敏感型系统，以基于文档数据模型的 NoSQL 系统作为典型案例展开研究，在本文中所处位置如图 3-1 所示。本章从网络和吞吐/内存敏感系统协同设计的角度，将提出一种 RDMA 增强的高性能、内存高效和高可用的文档数据模型，并在此基础上实现了对应的文档 NoSQL 系统 RMongo。

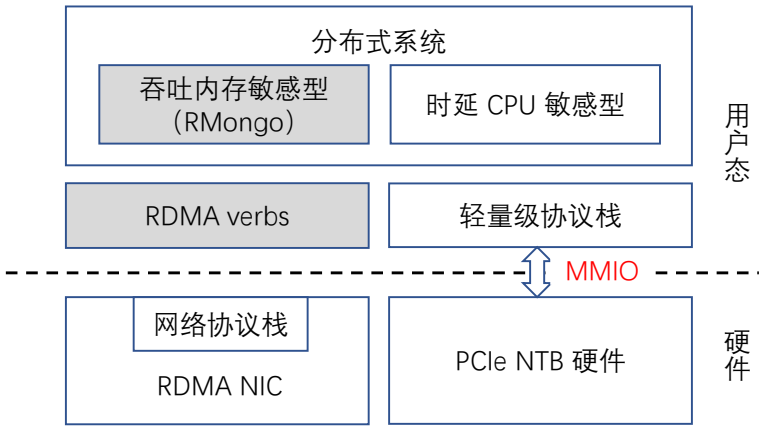


图 3-1　本章主题：基于 RDMA 增强吞吐/内存敏感型系统，用灰色背景的方框标记

3.1　引言

随着数据中心需要在线或离线收集和分析的大规模数据的迅速增加，MongoDB 等分布式 NoSQL（Not Only Structured Query Language）数据库系统在业界受到了广泛的部署和应用。NoSQL 数据库的设计初衷不仅是应对现代应用程序面临的扩展性和敏捷性两方面挑战，而且还要充分利用生产环境中大量可用的存储和计算资源。作为文档 NoSQL 系统的典型实现，MongoDB 将灵活的文档数据模型、自动分片、副本集作为关键特性，是当前支持多文档事务的代表性 NoSQL 系统。MongoDB 中的增删改查（Create/Update/Retrieval/Delete，CURD）文档操作、副本集和自动分片都依赖于底层网络通信。此外，大数据网络[77] 对于满足大数据处理的需求至关重要。在大数据处理高并发的情况下，我们在评测中发现，使用传统网络协议栈的传输负载已经成为 MongoDB 的性能瓶颈。

同时，作为近年来新兴的底层网络技术，RDMA（Remote Direct Memory Access），尤其是 RoCE（RDMA over Converged Ethernet），被很多工业数据中心频繁采用[10, 35–38, 78, 79]。它为 HDFS（Hadoop Distributed File System）[80, 81]、内存文件系统[58]、大数据分析和建模[82, 83]

等大数据应用程序提供理想的高吞吐量、低时延和 CPU 旁路。RDMA 网络协议使用零拷贝技术并绕过操作系统内核直接访问远程主机中的注册内存。

图 3-2 展示了 TCP 和 RDMA 之间的区别。在 TCP 方面，应用缓冲区中的数据在用户空间中被面向字节流的套接字接口封装，然后将封装后的数据通过网络套接字和操作系统内核的 TCP/IP 层协议写入网卡。但是，对于 RDMA 网络，应用缓冲区中的数据在用户空间中被面向内存语义的 RDMA 原语封装，封装后的数据直接绕过操作系统内核写入 RDMA 网卡。

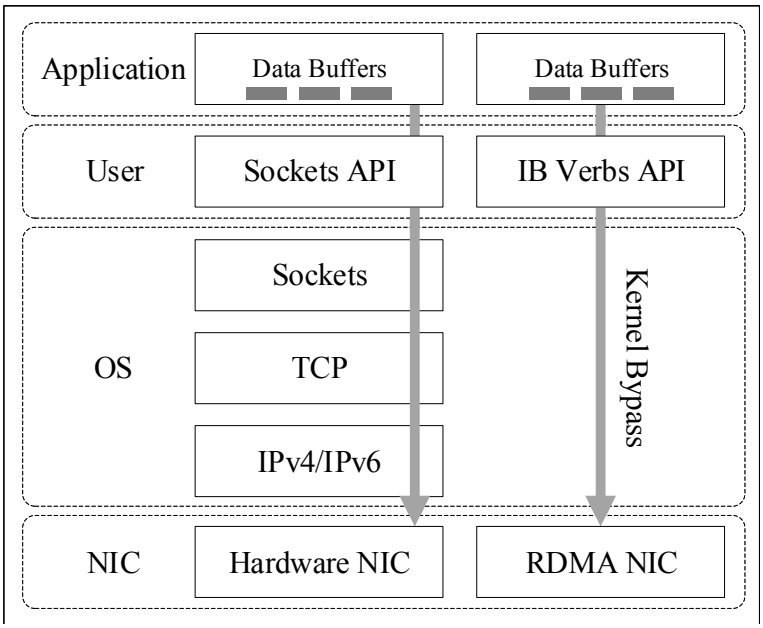


图 3-2　TCP 与 RDMA 协议栈的区别

RDMA 原语可以分为面向消息的双边原语（如 RDMA Send/Recv）和内存语义的单边原语（如 RDMA Read/Write）[17, 84]。消息传递通信要求接收方启动传输会话并注册一个通信缓冲区。之后，发送方准备修改远程接收方内存缓冲区中的数据并完成数据交换。然而问题在于，在这种面向消息的模式下，数据内存缓冲区和通信内存缓冲区之间的数据拷贝仍然是不可避免的。RDMA 内存语义的单边原语通过允许发送方和接收方保持完全被动，来合并应用数据内存和通信缓冲区，从而实现数据路径上的零拷贝。现有的 RDMA 驱动的数据中心应用程序，称为 RDMA 增强范式，分为两类：RDMA 增强系统[35–43, 45, 54, 57, 58, 80, 81, 85–89]和 RDMA 增强算法[90–93]。例如，Nessie[40] 通过内存语义的单边 RDMA 原语设计了一个完全由客户端驱动的高性能键值存储系统。APUS[91] 提出了第一个基于 RDMA 的 Paxos 共识算法。现有的 RDMA 驱动系统主要集中在分布式文件系统[58, 80, 81, 85]、键值存储[35–43]、并行数据库[57]、关系型数据库[54, 86]，以及分布式内存事务[87–89]。然而，据我们所知，还没有尝试在支持 RDMA 的网络中增强基于文档的 NoSQL 数据库。因此，我们能否通过 RDMA 加速基于文档的 NoSQL 系统以缓解传输瓶颈？

本文介绍了 RDMA_Mongo（RMongo），这是第一个 RDMA 驱动的基于文档的 NoSQL 范式。我们首先展示了对 MongoDB 网络传输设计的详细分析。通过分析高并发 NoSQL CURD 操作的关键阶段，我们发现有两个关键挑战：（1）减少传统网络中数据传输引起的频繁 CPU 上下文切换；（2）减少 MongoDB 客户端的等待时间。

表 3-1　近些年 RDMA 驱动的 NoSQL 数据库研究工作归纳分析

NoSQL 系统分类	RDMA 驱动的范式
键值数据模型	HydraDB [39]
	Nessie [40]
	InnerCache [43]
	HERD [38]
	Hybrid Memcached [35, 37]
图数据模型	Wukong [45]
文档数据模型	**RDMA_Mongo**

针对上述挑战，具体而言，RDMA_Mongo 会自动检测本地和远程主机是否支持 RDMA 网卡和相关库（如 libibverbs、librdmacm），然后确定是使用 RDMA 原语还是传统网络套接字。其次，根据当前系统负载（尤其是内存使用情况），RDMA_Mongo 按需确定合适的缓冲区大小并注册 RDMA 通信内存区域。第三，RDMA_Mongo 借助 RDMA 完成队列（CQ）的强大功能引入了非阻塞式的异步数据通信机制。RMongo 基于 RDMA 内存区域重新组织了文档数据模型的数据结构，并利用内存语义的 RDMA 单边原语扩展了文档数据模型的增删改查操作语义，以减少端到端数据路径上的内存拷贝次数和 CPU 上下文切换开销。最后，原生 MongoDB 利用副本集机制实现高可用，即 MongoDB 集群同时具有一个主节点和多个从节点。从节点需要不断地从主节点拉取操作日志（Operation Log，oplogs），并重放具有幂等性的操作日志进行数据库同步。为了加速这个过程，我们优化和重新设计了基于 RDMA 原语的操作日志同步协议。

我们工作的主要贡献可以总结如下：

- 我们彻底分析了 RDMA 驱动范式的所有各种设计选择，并展示了高性能 RDMA_Mongo 不同设计选项之间的有效权衡。
- 我们提出了一种 RDMA 上下文检测算法，用于确定两个 RDMA_Mongo 节点之间基于 TCP/IP 或 RDMA 的通信，并且还提出了负载感知缓冲区注册机制，以便对 RDMA 通道进行合理的内存区域管理。
- 我们在 RDMA_Mongo 中使用 RDMA 原语重新设计操作日志同步协议，包括优化操作日志初始同步和稳态复制，其中 RDMA 完成事件通道用于异步接收消息。
- 我们基于 MongoDB 4.1.1-59 版构建实现了 RDMA_Mongo。在基于 RDMA 的测试床中，实验评估表明，在面对大规模数据操作负载时，RDMA_Mongo 显著改进了 NoSQL 文档数据模型的增删改查操作性能，数据插入、更新、查询和删除操作的平均端到端吞吐分别提升了约 30%、17%、15% 和 30%。

在第 3.2 节，描述了典型的 NoSQL 类别、RDMA 网络特性和在 RDMA 增强范式方面的相关研究工作，然后介绍了研究动机。第 3.3 节展示了 RDMA_Mongo 架构的概述，并讨论了不同设计选项之间的权衡。在第 3.4 节中，提出了 RDMA 上下文检测算法和负载感知的 RDMA 缓冲区注册机制，并介绍了 RDMA 驱动的多分片节点之间操作日志同步机制设计。

在第 3.5 节中，我们评估了在不同的数据操作负载下 RDMA_Mongo 相比原生 MongoDB 的性能收益。第 3.6 节是本文的结论和我们对未来工作的展望。

3.2　背景与动机

3.2.1　吞吐/内存敏感的 NoSQL 系统与 RDMA 增强范式

1.　吞吐/内存敏感的 NoSQL 系统

吞吐/内存敏感的 NoSQL（Not only SQL）系统通常是指不同于关系数据库的面向多种数据模型的数据库系统[94]。随着数据处理规模的递增，传统的关系数据库很难提供高性能的数据管理服务。而吞吐/内存敏感的 NoSQL 数据库在开发人员中越来越受欢迎[95]。这个概念于 1998 年首次使用，并于 2009 年重新兴起[96]。NoSQL 数据库有四种主流数据模型驱动：键值（Key-Value）数据模型、面向列（Column）的数据模型、文档（Document）数据模型、图（Graph）数据模型[95, 96]。表格 3-1 展示了近年来 NoSQL 类别和 RDMA 驱动的范式。我们发现许多研究工作都关注在利用 RDMA 增强键值存储和图数据库系统。然而，关于 RDMA 驱动的文档 NoSQL 系统尚未有相关的研究。因此，我们首次提出了基于 RDMA 的文档模型 NoSQL 范式 **RDMA_Mongo**（即 RMongo）①。

键值数据模型：键值数据库使用一个键来匹配一个值，键值存储区维护一个包含一组记录的哈希映射或字典，不同的键可以通过识别和检索不同的数据记录。内存键值存储系统提供了的快速检索能力、高可伸缩性和并发性。但是，复杂的条件查询在 Redis、Memcached 等键值存储的端到端性能较差。

面向列的数据模型：在面向列的数据模型中，数据元素按照列索引进行数据的高效组织和访存。该数据模型的优点包括：1）极高的加载速度；2）适合大量数据；3）高效的压缩率；4）更适合聚合和数据仓库的应用。但它不适用于扫描小数据、随机更新、实时删除和更新[97]。HBase[98] 是一个常用的面向列的数据库。

文档数据模型：面向文档数据模型的数据结构与键值数据库类似，但在大规模数据查询负载下能实现更高的数据元素索引效率。面向文档的数据库存储半结构化的文档数据结构，并以特定的格式编码数据，包括 JSON、XML 和 YAML。此外，面向文档数据模型的 NoSQL 系统中还支持一个辅助索引，以满足更高效的查询需求。而 MongoDB[9] 是业界普遍应用的一个典型文档 NoSQL 系统。

图数据模型：图数据库利用灵活的图结构来实现更高的可伸缩性和更快的语义查询。图状数据库使用节点、边和属性来表示和存储数据项。不同数据实体之间的关系被抽象为边，这显著提升了面向存在关联的数据集合的查询性能。图模型可分为带有标签的属性图、资源描述框架（Resource Description Framework，RDF）两类。典型的图数据库包括 Neo4J[99] 和 Infinite Graph.。Wukong[45] 提出了 RDMA 友好的 RDF 图查询平台。

① 在 2019 年 6 月 RMongo 工作录用时，据我们调研所知，尚未存在基于 RDMA 的 NoSQL 文档数据模型的相关研究工作。

2. RDMA 增强范式

新兴的 RDMA 技术通过零拷贝和绕过远程操作系统内核提供超低时延和高吞吐量。它通过内核旁路网络将数据直接写入预先注册的远程内存区域，并将数据从本地内存移动到远程内存，而无需任何 CPU 参与。它消除了数据路径上多余的内存拷贝次数和 CPU 上下文的切换开销。

零拷贝技术允许 RDMA 网卡 (RNIC) 直接利用注册的 RDMA 内存区域进行数据的端到端传输，避免用户空间和内核空间之间、内核空间和 RNIC 之间的数据拷贝开销。通过绕过操作系统内核和远端 CPU，基于 RDMA 传输的数据载荷首先从用户空间的应用进程直接发送到本地 RDMA 网卡硬件，再通过基于优先级流控的无损以太网将数据包发送到远端网卡硬件，在网卡内完成数据包的协议栈处理，从而使得内核空间与用户空间之间的 CPU 上下文切换次数显著地降低。

RDMA 上有两种类型的通信原语：内存语义的单边 Write/Read 原语、消息通道语义的双边 Send/Recv 原语。单边 RDMA Write/Read 可以直接从本地内存写入远程内存，而无需涉及 OS 内核，而双边 RDMA Send/Recv 数据路径上需要两端 CPU 的参与，单边 RDMA 原语比双边 RDMA 快 2 倍左右。本地网卡和远程网卡之间的每条连接都会维护一个 QP（Queue Pair）对象，其中包含一个发送队列和接收队列。递增的并发连接数量意味着对应的 RDMA 队列对数量的增多。每个 QP 可以维护一个 CQ（Completion Queue），采用一个先进先出 (First Input First Output，FIFO) 队列模型，存储所有已完成的工作请求（Work Request，WR）项。每个完成的 WR 会对应生成一个完成队列条目（Completion Queue Element，CQE）并发布到对应的完成队列（Completion Queue，CQ）。通常，一个 CQE 在被异步的守护进程从队列头部轮询和回调处理后，就会从完成队列中移除。

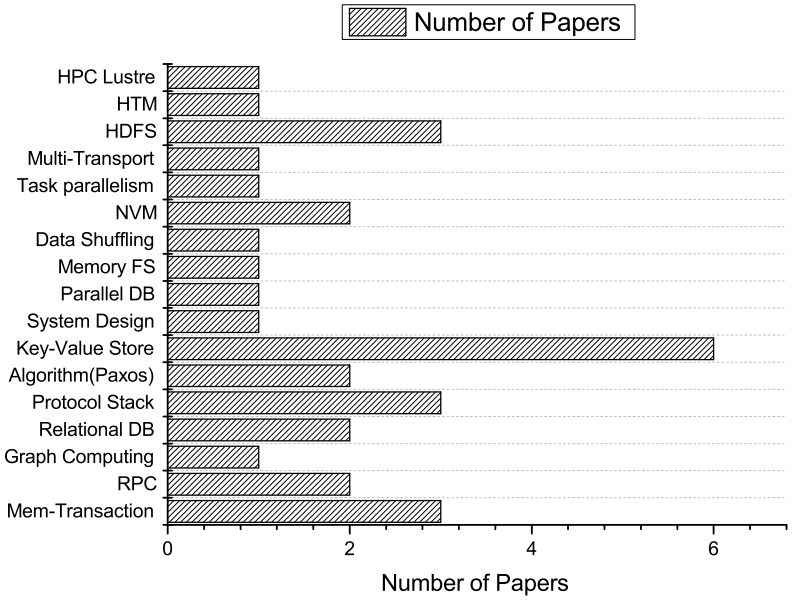


图 3-3　近些年在系统顶级会议或期刊上高性能 RDMA 相关的研究话题

图 3-3 展示了近年来许多探索 RDMA 功能的新兴研究工作。先前的工作大致可以分为两类：RDMA 增强系统[35–43, 45, 54, 57, 58, 80, 81, 85–89] 和 RDMA 增强算法[90–93]。

对于 RDMA 增强型系统，大数据分布式文件系统[58, 80, 81, 85]，键值存储[35–43]，并行数据库[57]，关系数据库[54, 86]、图计算[45]、RPC 库[87, 100]、分布式内存事务[87–89] 都采用 RDMA 来实现高性能。Pilaf[42] 通过单边 RDMA Read 原语实现高性能键值存储，同时提出自验证数据结构以保证并发内存修改下的缓存一致性。Octopus[58] 利用 RDMA 与非易失性内存硬件的协同设计，提出了面向新型网络和存储硬件特性的数据/元数据操作机制，显著地增强了内存文件系统的端到端文件访存性能。Wukong[45] 是一种 RDMA 增强的分布式内存 RDF 图存储，它提出了 RDMA 驱动的图探索机制，从而提供面向大规模图数据的内存高效、更高吞吐和低时延的 RDF 图查询性能。DrTM+H[89] 提出了一种分布式内存事务系统，该系统以多种 RDMA 原语混合协同的方式，实现了每个事务处理阶段的最佳性能收益。

对于 RDMA 增强算法，多任务调度[92, 93] 和共识算法[90, 91] 等算法已在支持 RDMA 的网络上进行了优化。Uni-Address[93] 是一种基于 RDMA 的线程管理方案，具有可扩展的工作窃取功能，用于高性能分布式多任务调度。DARE[90] 利用 RDMA 原语通过无等待日志拷贝来加速状态机复制协议，从而实现更高的请求率和更低的写/读请求时延。APUS[91] 使用 RDMA 原语替换 Paxos 协议中的传统 TCP/IP，以降低共识时延，其性能比 DARE[90] 高 4.9 倍。

3.2.2　动机

在许多数据中心中，大数据处理平台依赖于文档 NoSQL 系统进行底层数据存储和迁移。然而，传统文档 NoSQL 系统内基于传统以太网的数据传输引入了过多的 CPU 开销、无法容忍的时延和吞吐量开销，这很难满足在线事务处理的网络需求。此外，现有的 RDMA 增强型 NoSQL 系统主要是键值存储[35–43]、面向列的存储[101] 和基于图的 RDF 存储[45]，在面向商用 RDMA 网络的文档数据模型性能增强方面仍然欠缺系统性的探索和研究。因此，受新兴网络硬件 RDMA 增强范式的启发，我们尝试面向基于文档数据模型的 NoSQL 系统利用 RDMA 技术来加速系统的数据传输和操作日志同步机制，从而在没有 CPU 干预的情况下实现更高的端到端吞吐和更低的时延。

3.3　RMongo 系统概览

RDMA_Mongo 是基于文档的 MongoDB 的 RDMA 感知设计，同时有效地利用了 MongoDB 传输层的 RDMA 设计空间。在本节中，我们首先介绍了 RDMA_Mongo 的系统概述，包括系统架构和不同组件之间涉及 RDMA 的关键通信。然后，我们展示了 RDMA 增强系统优化的整体设计空间。最后，我们讨论了不同设计选择之间的权衡，以实现更高的吞吐量和更低的时延。

3.3.1　整体架构

RDMA_Mongo 使用 RDMA 原语替换传输层，同时保持原始架构不变。图 3-4 展示了 RDMA_Mongo 架构。RDMA_Mongo 涉及三个关键组件：作为路由器的 mongos、作为配置服务器的 mongod 和服务于数据高可用的分片节点。在这些组件中，数据传输负载比较大的

主要有 3 个阶段，包括 client 和 mongos 之间的 CURD 操作，mongos 和 shard 集群之间的数据读写，shard 节点之间的 oplog 同步。RDMA_Mongo 采用 RDMA 感知的传输层来加速上述三个关键传输阶段。

接下来，我们深入分析基于 RDMA 增强文档数据模型的设计空间，并详细介绍了不同的设计选项之间的权衡与协同设计，从而获得面向商用 RDMA 网络的高性能文档数据模型。

3.3.2　RDMA 驱动的范式

图 3-5 展示了 RDMA 可以工作在三种类型的网络上：Infiniband、RoCE（包括 RoCEv1 和 RoCEv2 两个版本）和 iWARP。在用户空间 API 层，所有范式都基于 IB 原语 API。开发者可以使用一些库基于 iSER、rsocket、MPI、SDP 等 IB 原语，或者直接使用 IB Verbs API 开发 RDMA 驱动的应用。虽然这些库为开发提供了极大的便利，但由于这些库导致的 socket-to-Verbs 转换的高开销，性能将严重下降。因此，为了达到最佳性能，我们直接使用 IB 原语 API 来开发我们的 RDMA_Mongo，而不使用任何第三方库。

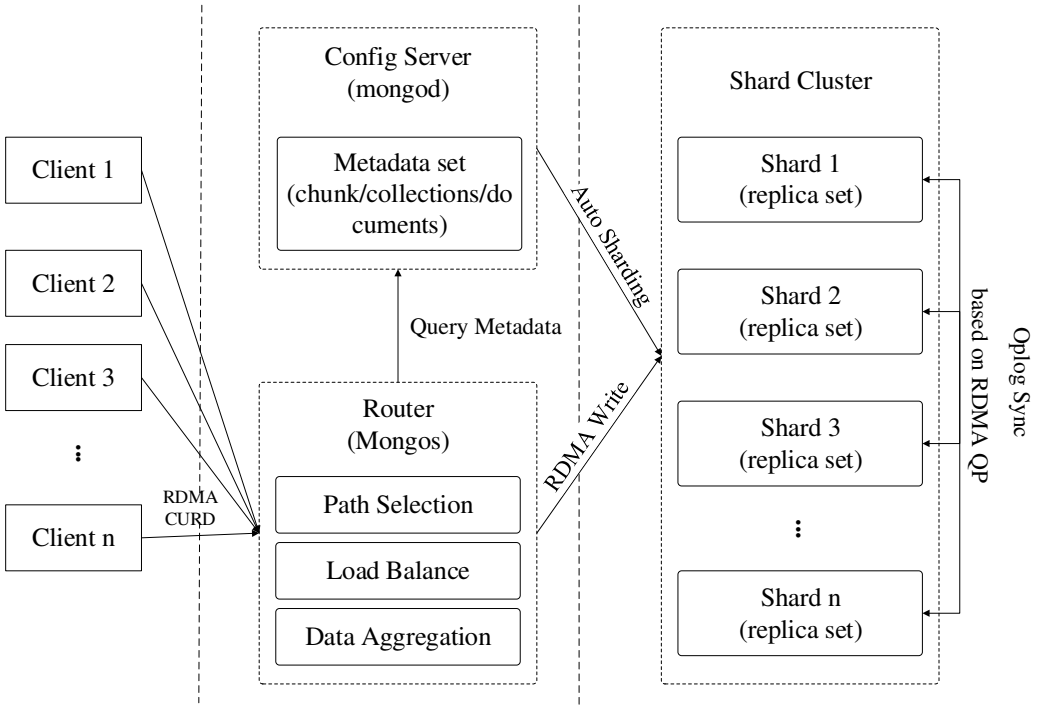


图 3-4　RMongo 整体架构

3.3.3　设计空间

RDMA 典型的数据传输工作流程包括 RDMA 设备初始化、队列对 QP 的创建、RDMA 内存空间注册、连接级 QP 元数据交换、基于 RDMA Send/Recv/Write/Read 原语的数据收发、RDMA 实体资源的释放。队列对 QP 由一个发送队列和一个接收队列组成，并与完成队列相关联。工作请求 WR 必须异步发布到 QP 中以进行通信。一旦工作请求被网卡硬件内固化

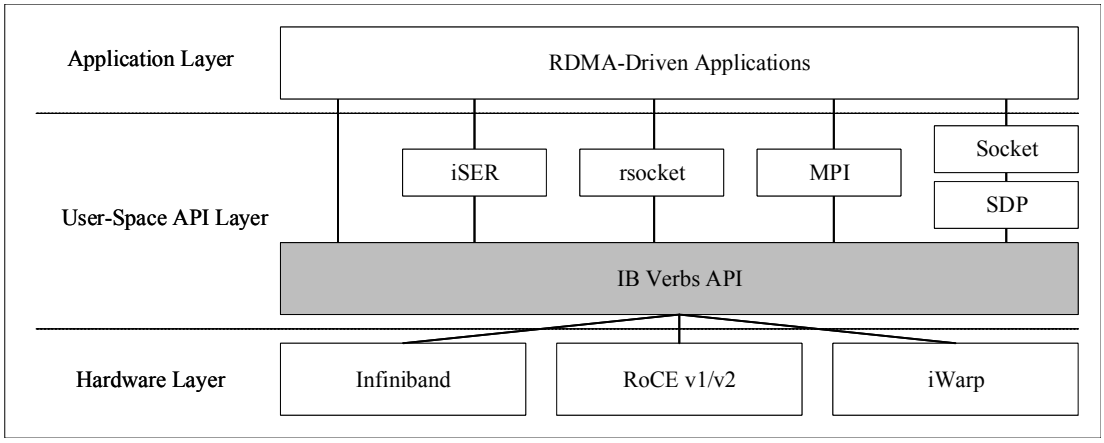


图 3-5　RDMA 驱动范式

的协议栈解析处理，一个完成事件将被生成并推送到该 QP 连接对应的完成队列。然后应用程序从完成队列中轮询完成事件以安全地复用内存区域。相关的设计空间主要可以归纳为以下几个方面。

RDMA 传输类型和功能： RDMA 有两种传输服务类型，包括可靠连接（Reliable Connection，RC）和不可靠数据报（Unreliable Datagram，UD）。传输功能包括面向消息的双边 RDMA Send/Recv 原语和内存语义的单边 RDMA Read/Write 原语。在面向连接的 RDMA 可靠连接 RC 模式下，n 个节点之间的端到端传输需要 n^2 个 QPs。在无连接的 RDMA 不可靠数据报 UD 模式下，n 个节点之间的端到端传输只需要 n 个 QPs。RC 服务支持单边和双边通信，而 UD 服务只支持双边通信。

RDMA 通信的消息大小： 在 UD 传输服务中，MTU 是消息大小的上限。在 RC 传输服务中，Infiniband NIC 中的最大消息大小可以为 2 GB。较小的消息大小会导致更多的工作请求，从而在传输过程中带来更多的 CPU 开销。但是，更大的消息大小需要更多的注册内存用于 RDMA 通信，这会导致更高的内存消耗。因此，合适的消息大小对于 RDMA 传输是必不可少的。

每个主机的队列对数： 每个主机节点所需的 RDMA 队列对数量需要在传输并行性和 RDMA 硬件网卡缓存资源开销之间进行设计权衡。先前的工作[44] 已经证明，每个节点过多的 QP 会导致性能下降多达 5 倍，并且很容易在较大的集群中使 NIC 片上缓存溢出。但是，在高并发下，每个节点的 QP 太少会导致线程争用。在 RDMA 可靠连接 RC 模式下，QP 的数量必须等于连接的数量，并受到 RNIC 硬件缓存资源的严格限制。在不可靠数据报模式下，最佳实践是保持 QP 的数量等于 CPU 内核的数量。

3.3.4　设计选项权衡

RDMA 双边原语与单边原语： RDMA 三种传输模式在数据路径上的数据包处理上具有不同的资源开销。由于没有确认数据包，不可靠数据报传输模式需要应用程序处理乱序数据包和错误，这会导致更多的 CPU 参与。由于每个传输的数据包都需要 RDMA 网卡中的确认数据包，因此可靠连接传输服务类型通过更简单的算法实现了高吞吐量。在流量控制方面，RDMA 双边原语（例如 RDMA Send/Recv 原语）要求应用程序对传输的消息进行计数，

并持续跟踪发送方和接收方之间发布的发送/接收请求的数量。因此，这会导致频繁的 CPU 上下文切换开销，以及通信内存缓冲区和数据内存缓冲区之间的多次数据拷贝开销。同时，当接收方的消息缓冲区可以安全地释放用于传入覆盖时，单边 RDMA 原语只需要通知发送方。因此，本文采用了更直观的具有可靠连接的单边 RDMA 原语。

RDMA 内存固定大小分配与按需分配：RDMA 内存的固定消息大小分配很难适应动态系统负载。考虑到内存消耗和当前网络吞吐量之间的平衡，亟需一个面向 RDMA 传输的负载感知的按需分配机制。

扩展与保留原始接口：通过 RDMA 原语扩展 MongoDB 传输层接口导致上层调用的修改成本很高。因此，在原始传输接口上没有变化的情况下，RDMA_Mongo 用 RDMA 单边 Write/Read 原语替换套接字实现，它可以与传统的 TCP/IP 网络堆栈共存。

3.4　系统设计与实现

本节详细介绍了 RDMA_Mongo 使用的基于 RDMA 的上下文检测、缓冲区注册策略、原语选择和操作日志同步。RDMA 上下文检测和负载感知缓冲区注册算法分别在初始化和连接建立阶段执行。RDMA_Mongo 操作日志同步协议包含了面向辅助节点的 RDMA 驱动的操作日志获取算法、面向主节点的 RDMA 驱动的操作日志监测算法。RDMA_Mongo 采用的上述四种算法之间的关系如图 3-6 所示。

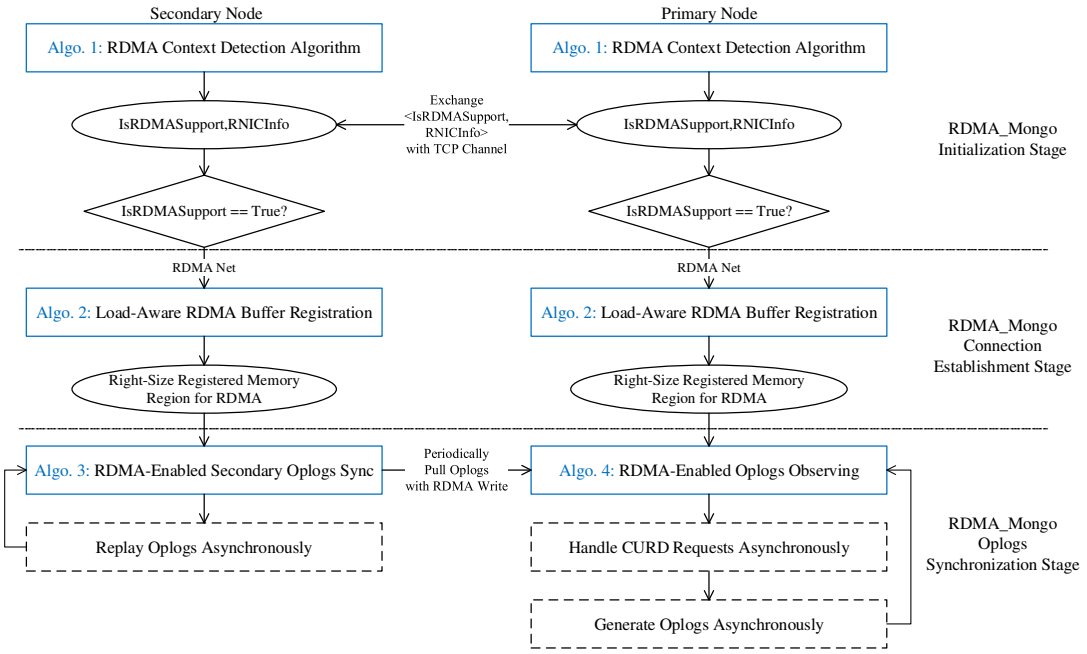


图 3-6　RDMA_Mongo 提出的算法 1、算法 2、算法 3 和算法 4 之间的协同关系

3.4.1　RDMA 上下文检测算法

在支持 RDMA 的传输中，我们应该确保客户端和服务端机器都支持互连且兼容的 RDMA 硬件。RDMA_Mongo 配置文件 *mongod.conf* 中新增了一个名为 *is_rdma* 的配置项，其中

true 表示支持 RDMA 硬件，*false* 表示不支持 RDMA 硬件。在 RDMA_Mongo 初始化过程中，本地节点加载 *mongod.conf* 文件以提取用户定义的配置项。如果 *is_rdma* 项为真，本地节点将通过 *ibv_get_device_list()*/*ibv_get_device_name()* 查询本地可用的 IB 设备名称，通过 *ibv_query_port()* 查询活动设备端口，通过 *show_gids* 命令查询 *gid*（RDMA 全局地址）索引和有界 IP。之后，会在本地和远程节点之间建立一个 TCP 通道，将本地 *is_rdma* 和 RNIC 元数据同步到远程节点，同时远程 RNIC 元数据将返回本地节点。如果本地和远程节点的 *is_rdma* 的值都为真，则将建立两端之间的 RDMA 通道进行数据传输。否则 TCP 通道用于传输数据。算法 1 展示了前面提到的 RDMA 上下文检测算法。

Algorithm 1 RDMA 上下文检测算法

Require:

None

Ensure:

bool isRdmaSupport

struct remoteRnicInfo

local_support, remote_support ⇐ *Null*
localRnicInfo, remoteRnicInfo ⇐ *Null*
if local node rdma status is *True* **then**
    local_support ⇐ True
    localRnicInfo.dev_name ⇐ *local IB device name*
    localRnicInfo.ib_port ⇐ *local IB device port*
    localRnicInfo.gid_idx ⇐ *local IB device gid index*
    localRnicInfo.ip ⇐ *local IB device bounded IP*
else
    local_support ⇐ *False*
end if
tcpSyncData(*local_support*, *remote_support*, *localRnicInfo*, *remoteRnicInfo*)
if *local_support* and *remote_support* are *True* **then**
    isRdmaSupport ⇐ *True*
    return *True*
else
    isRdmaSupport ⇐ *Flase*
    return *False*
end if

3.4.2　负载感知的缓冲区注册算法

对于 RDMA_Mongo 高效传输，当前内存使用和网络负载是至关重要的影响因素。我们展示了一种适用于 RDMA 通信内存区域的自适应缓冲区注册算法，如算法 2 所示。在有

足够的可用内存和网络的情况下需要稍大的缓冲区大小，而在空闲内存和网络稀缺的情况下需要较小的缓冲区大小。本文将空闲内存率和 RNIC 吞吐率的乘积作为过载因子来表示资源的稀缺性。当这个过载因子小于指定的过载阈值 *overloadThr* 时，我们将负载标记为高，并根据公式 3.1 计算缓冲区大小。否则，我们将负载标记为低并使用一个基线值作为缓冲区大小。我们根据经验确定基线缓冲区大小 S[85]。公式 3.1 展示了上述算法的核心思想，其中常数 k 为正则因子。

$$M = k \times S \times (1 - \frac{THR}{B}) \times (1 - \frac{U}{T}) \tag{3.1}$$

其中 M 是计算出的 RDMA 内存缓冲区大小，正则因子 $k \in (0, 1)$，可以用于控制 RMongo 系统所占内存资源的比例，避免过度抢占其他应用进程的可用内存资源，S 是由 RMongo 操作日志大小决定的基线缓冲区大小，THR 是当前网络吞吐量，B 是 RNIC 的最大带宽限制，U 是当前被占用的内存大小，T 是总主机内存大小。Oplog 操作日志大小可以在 RDMA_Mongo 配置文件中指定。本章实验评估中所使用的 k、S、B 和 T 的实际值分别为 *0.7*、*50MB*、*100Gbps* 和 *32GB*。

Algorithm 2 负载感知的 RDMA 缓冲区注册算法

Require:

float baseBuffer

float overloadThr

float k

Ensure:

float bufferSize

```
memUsage ⇐ current memory usage in host
netUsage ⇐ current network throughput in host
calculate freeMemRate and freeNetRate
loadFactor ⇐ freeMemRate * freeNetRate
if loadFactor in range (0, overloadThr) then
    bufferSize ⇐ baseBuffer * loadFactor * k
else
    bufferSize ⇐ baseBuffer
end if
```

3.4.3　可靠连接模式下的 RDMA Write

当提到 RDMA 单边通信原语时，发送方和接收方端点之间的协调机制是一项重大挑战。我们以 RDMA_Mongo 客户端和 mongos 组件之间基于 RDMA 的 CURD 请求/响应处理为例。

消息可以分为两类：控制消息和数据消息。具有恒定大小（小于 4KB）的控制消息通常在小的注册缓冲区上交换，而具有高度可变大小的数据消息通常需要大的注册内存缓冲区，不同类型的请求-响应消息根据文档数据模型的通信协议，在数据包头设置了不同的消息类

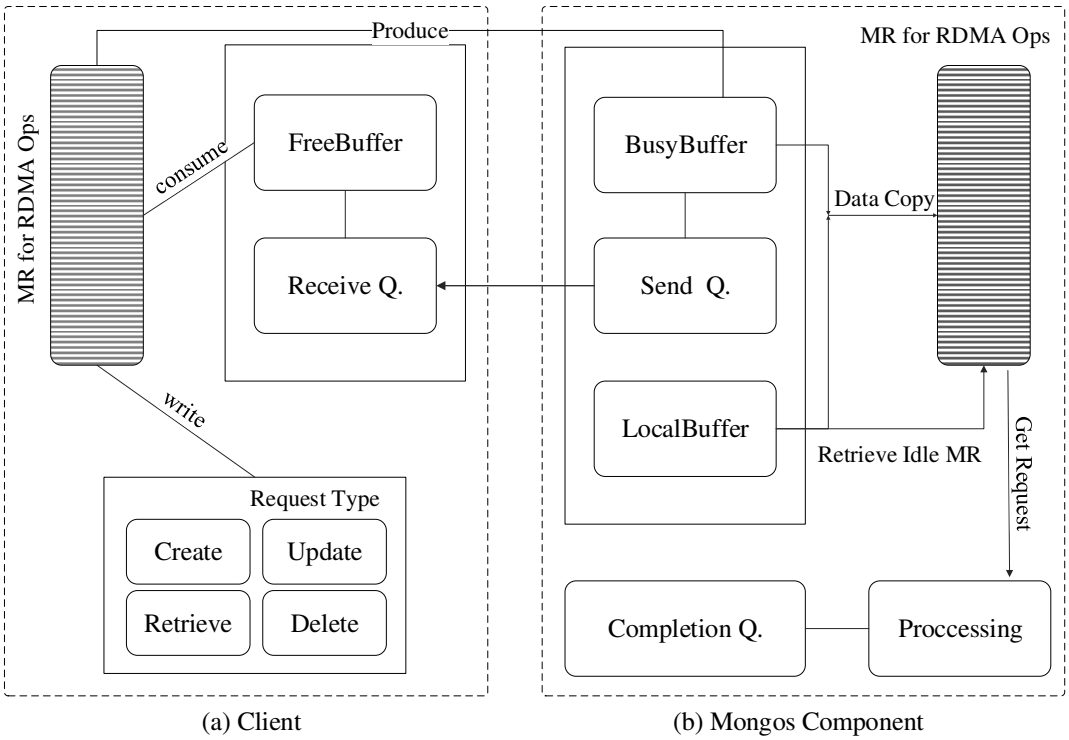


图 3-7　RDMA_Mongo 中 Client 客户端与 Mongos 组件之间的协同机制

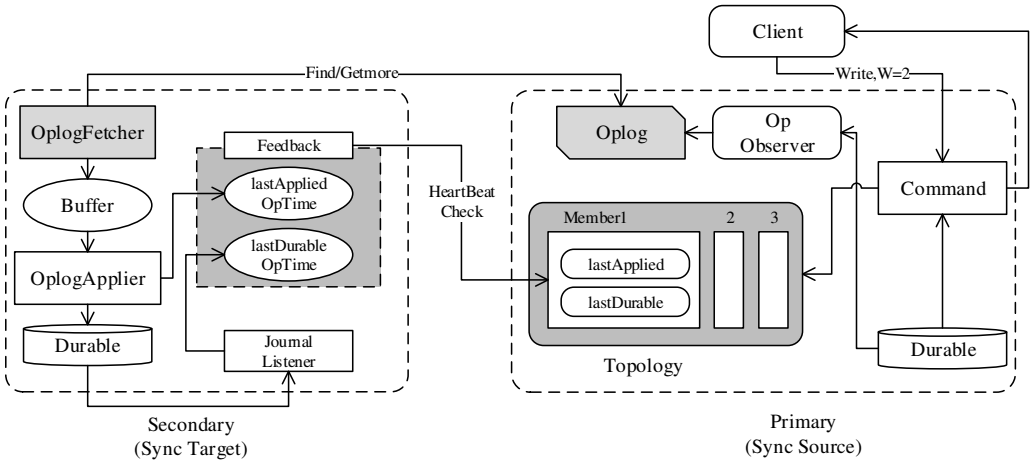


图 3-8　RDMA_Mongo Oplogs 数据同步机制

型。对于 RDMA_Mongo 客户端作为控制消息发送的 CURD 请求，需要在 mongos 组件上注册小的 MR 内存区域作为接收缓冲区。对于 mongos 组件发出的数据插入 *Create*、数据更新 *Update*、数据删除 *Delete* 响应，客户端上需要小的接收缓冲区。而对于 mongos 组件作为数据消息发出的查询响应，需要在客户端注册大量的接收缓冲区，并且需要提前在查询请求中包含相应的内存地址和远程密钥（remote key, rkey）。客户端和 mongos 之间的每单边都包含一个小的预注册发送/接收缓冲区，用于发送与接收控制消息。考虑到发送与接收缓冲区争用，RDMA_Mongo 应该识别发送/接收缓冲区何时空闲以被消耗或重用，以及发送与接收缓冲区何时被占用。

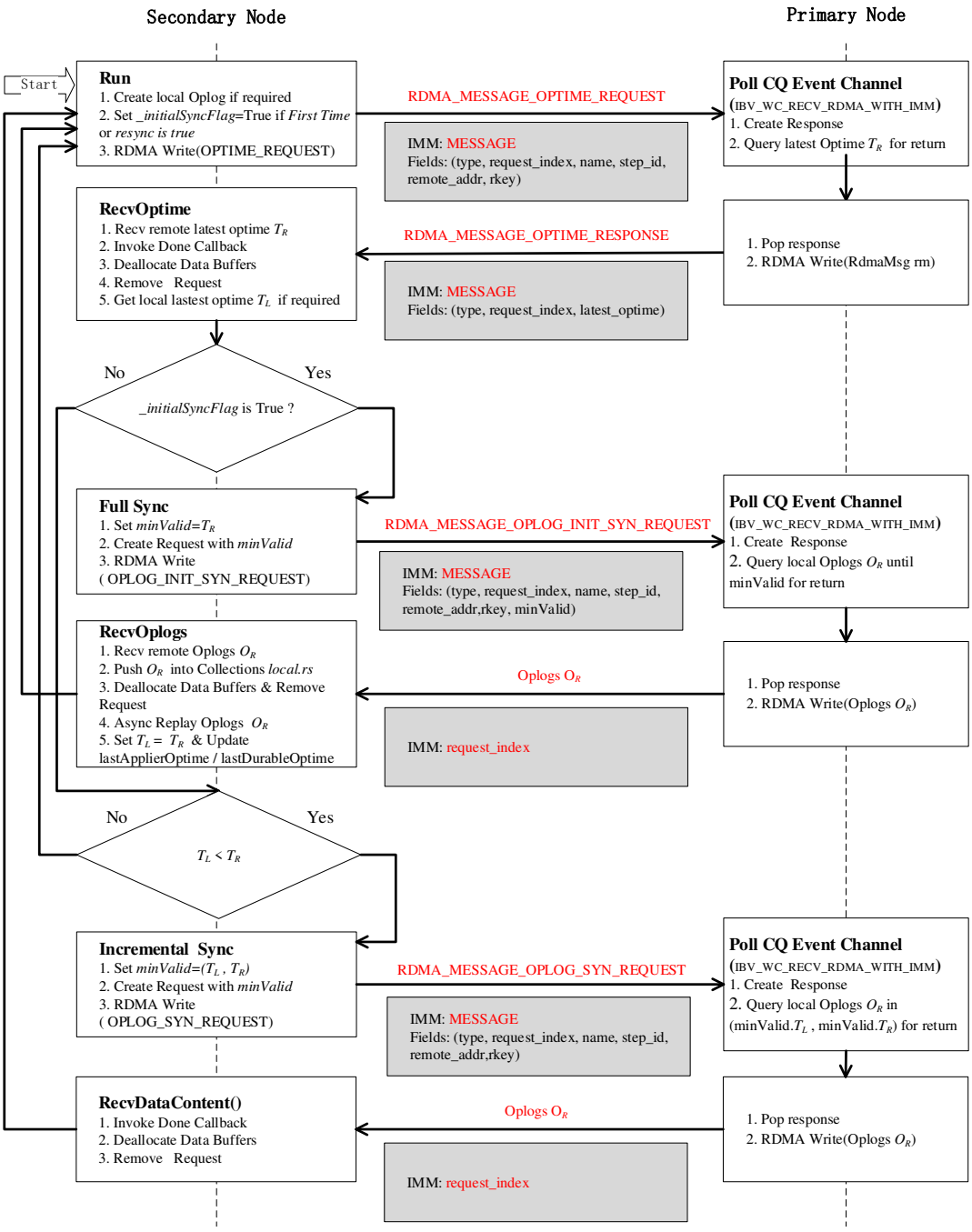


图 3-9　RDMA_Mongo 中基于 RDMA 的 Oplogs 操作日志数据同步协议

图 3-7 展示了客户端和 mongos 之间基于 RDMA 的数据增删改查 CURD 请求-响应处理机制的详细协议过程，其中包含 RC（可靠连接）服务中的单边 RDMA 写原语（one-sided RDMA Write）。注册的缓冲区被协调机制标记为空闲 *idle* 或忙碌 *busy*，以响应环形缓冲区循环队列 *freeBuffer* 和 *busyBuffer*。*free Buffer* 和 *busyBuffer* 中的条目分别对应 RDMA_Mongo client 和 mongos 组件维护的空闲 idle 和被占用 busy 注册缓冲区。RDMA_Mongo 客户端在发出 CURD 请求之前，从 *freeBuffer* 队列中获得一个空闲的预注册缓冲区，并将相关的地址和对应 *rkey* 封装到 CURD 请求消息中。客户端使用带有即时数据的 RDMA 写入将数据的增删改查 CURD 请求写入到 mongos 上的远程预注册控制消息缓冲区。同时，当 mongos 中的

完成事件通道检测到新的完成任务时，会触发一个事件，通知 mongos 轮询完成队列（CQ）。

之后，mongos 从 *busyBuffer* 读取增删改查 CURD 请求，提取对端地址和 *rkey*，处理请求，将繁忙的接收缓冲区释放到空闲状态并生成响应。然后使用 RDMA_Mongo 客户端提供的 *rkey* 将响应消息写入对端地址。客户端中接收响应的过程与 mongos 组件中接收请求的过程类似。

3.4.4　RDMA 驱动的操作日志同步机制

为了实现高可用和冗余，MongoDB 引入了副本集机制来维护相同的数据集，其中包含一个主节点和多个从节点。辅助节点不是直接从主节点拉取数据集合，而是获取远程操作日志（oplog）并重放这些 oplog 以实现数据持久化，如图 3-8 所示。Oplogs 数据同步包括两个阶段，即初始同步（*initial sync*）和稳态复制（*steady state replication*）。初始同步是耗时的全面同步，而稳态复制速度更快，增量获取更频繁。我们主要关注这两个阶段中使用 RDMA 原语的通信协议的优化和重新设计。有三种情况可以触发初始同步，包括新节点刚加入时、上次初始同步失败时以及触发 *resync* 命令时。

一旦确定了目标同步源节点，从节点将启动一个 *OplogFetcher* 线程，不断轮询远程最新的 oplog 时间戳 *TR*，将 *TR* 与本地最新的 oplog 时间戳 *TL* 进行比较。布尔变量 *_initialSyncFlag* 标识初始同步阶段。如果 *_initialSyncFlag* 为真，*OplogFetcher* 将获取远程 oplog，直到名为 *minValid* 的一致点 *TL*。如果 *TL* 小于 *TR*，则 *OplogFetcher* 以时间戳间隔 （*TL, TR*）获取远程 oplog，这意味着 *minValid* 为 *TR*。然后 *OplogApplier* 线程负责将获取的 oplog 重放到本地持久化中。主节点周期性地向从节点发送心跳包，以获取上次应用/持久 oplogs 时间戳并更新本地拓扑状态。

Algorithm 3 RDMA 驱动的副本节点 Oplogs 数据同步机制

Require: *bool* resync, *int* timeout

resync: bool flag for restarting initial sync
rn: remote target source node for oplogs sync
opq: oplogs blocking queue for caching fetched oplogs
timeout: sleep time for starting the next oplogs sync
wc: pre-allocated work completions array used for polling
MAX_CQ_NUM: max number of CQs

Ensure: *None*

_initialSyncFlag ⇐*bool* flag for starting initial sync
_pause ⇐*bool* flag for pause sync process
_shutdown ⇐*bool* flag for shut down sync process
if T_L is *null* or *resync* is *true* **then**
    _initialSyncFlag ⇐ *true*
end if
while *_shudown* is false **do**
    if *_pause* is true **then**

```
        break
    end if
    T_R ⇐ requestRdmaWriteRemoteOptime(rn)
    if _initialSyncFlag is true then
        ops ⇐ requestRdmaWriteSyncOplogs(rn, T_R)
    else
        if T_L < T_R then
            ops ⇐ requestRdmaWriteSyncOplogs(rn, T_L , T_R)
        else
            sleep(timeout)
            continue
        end if
    end if
    enqueueOpQueue(opq, ops)
    asyncReplayOplogsFromQueue()
    T_L ⇐ T_R
    updateLastApplierOptime()
    updateLastDurableOptime()
end while
```

为了将上述传输任务彻底卸载到 RDMA 网卡中，我们利用具有即时数据的内核旁路 RDMA Write 原语来发送消息和操作日志。图 3-9 具体描述了 RDMA 驱动的操作日志同步机制。RDMA CQ（Completion Queue，CQ）事件通道被用来异步监听传入的 RDMA 消息请求。为了区分不同的请求-响应并更好地兼容支持 RDMA 的通信，我们定义了一个 RDMA 消息数据结构，如表格 3-2 所示。算法 3 展示了辅助节点中 RDMA 驱动的副本节点 Oplogs 数据同步机制，算法 4 展示了 RDMA 驱动的 Oplog 数据监测节点。

表 3-2　RDMA 消息结构定义

Field	Size	Field	Size
type	1B	rkey	4B
name_size	2B	data_type	XB
name	512	oplogs_shape	XB
step_id	8B	oplogs_bytes	8B
request_index	8B	min_valid	8B
remote_addr	8B	error_status	size = 4B
checksum			proto = XB

Algorithm 4 RDMA 驱动的 Oplog 数据监测机制

Require:

cq: RDMA completion queue

ec: RDMA completion event channel

ct: RDMA context entity

Ensure: *None*

1: *_isRun* ⇐ *True*
2: *_cc* ⇐ *CQ context*
3: *_cq* ⇐ *extracted CQ from CQ event channel*
4: **while** *_isRun* is *True* **do**
5: ibvGetCqEvent(*ec*, *_cq*, *_cc*)
6: **if** *cq* != *_cq* **then**
7: continue
8: **end if**
9: ibvAckCqEvents(*_cq*, 1)
10: ibvReqNotifyCq(*_cq*)
11: *ne* ⇐ ibvPollCq(*cq, MAX_CQ_NUM * 2, wc*)
12: **if** *ne* == 0 **then**
13: continue
14: **end if**
15: **for all** WorkCompletion, $wc_i \in wc$ **do**
16: **if** wc_i.*status* is not *IBV_WC_SUCCESS* **then**
17: continue
18: **end if**
19: **if** wc_i.*opcode* is *IBV_WC_RECV_RDMA_WITH_IMM* **then**
20: *imm* ⇐ wc_i.*imm*
21: doRecvImmCallback(*imm,* wc_i)
22: **if** *imm* is *OPTIME_REQUEST* **then**
23: T_R ⇐ queryLastOptime()
24: rdmaWriteResponse(T_R)
25: continue
26: **else if** *imm* is *OPLOG_SYN_REQUEST* **then**
27: O_R ⇐ queryOplogs(wc_i)
28: rdmaWriteOplogs(O_R)
29: **end if**
30: continue
31: **else if** wc_i.*opcode* is *IBV_WC_RDMA_WRITE* **then**
32: doRdmaWriteCallback(wc_i)
33: **end if**
34: **end for**
35: **end while**

3.5　系统性能评估

在本节中，我们从两个维度评估 RDMA_Mongo 在 RoCE 网卡上的吞吐量、时延和消耗时间：单线程性能（第 3.5.2 节）和多线程性能（第 3.5.3 节）。在每个维度中，我们评估面向文档数据模型的插入、删除、更新和查询四种操作的端到端性能。第 3.5.1 节描述了实验测试床的软硬件配置、关键性能指标和参数取值。第 3.5.2 节展示了在文档数据模型的单线程工作负载下文档 NoSQL 系统基本增删改查数据操作的性能表现与实验分析。在第 3.5.3 节中，我们评估分析了多线程并发负载下文档数据模型增删改查操作的端到端时延。

3.5.1　实验设置

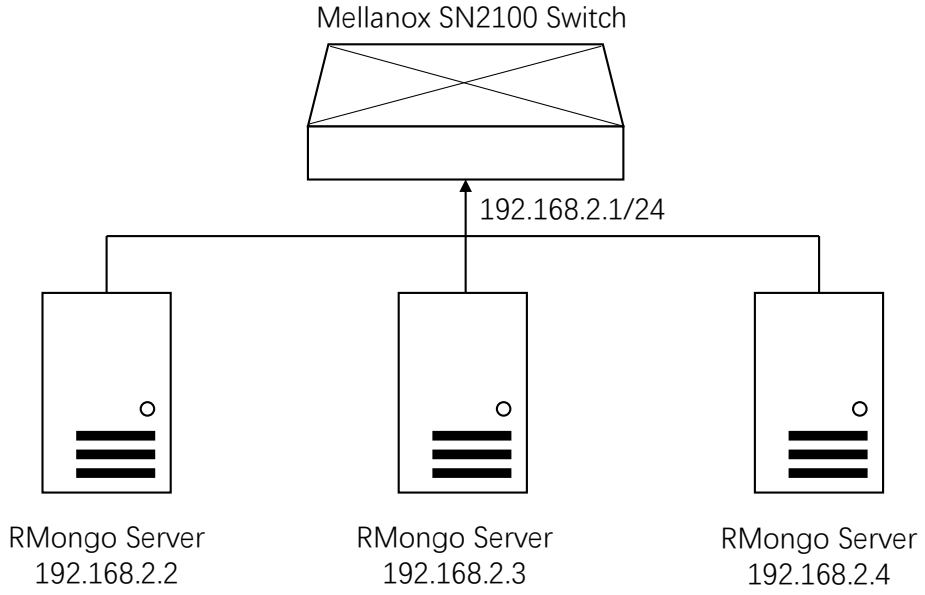


图 3-10　测试床的网络拓扑

性能实验在三台服务器组成的本地 RDMA 集群上进行，由 Mallanox SN2100 交换机互连，如图 3-10 所示。我们的测试平台运行在 RoCEv2 网络上。每台服务器都有一个 32GB DRAM 和两个频率为 3.00 GHz 的 12 核 Intel Xeon E5-2687W v4 CPU 处理器。每个 CPU 核都配备了一个的 768KB L1 缓存和一个的 3072KB L2 缓存。一个 CPU 处理器上的 10 个内核共享一个 30MB 的 L3 缓存。每台服务器都配备了一个 onnectX-5 MCX516A-CDAT 100Gbps RoCE NIC，通过 PCIe 3.0×16 连接到一个 Mallanox SN2100 100Gbps RoCE 交换机。CentOS 7.4.1708 和 OFED 4.5-1.0.1 堆栈在每台服务器上运行。RDMA_Mongo 原型系统基于 MongoDB 4.1.1-59-g1dd056d 发行版构建。

对于 RDMA_Mongo 和 TCP MongoDB 之间的性能比较，我们专注于三个重要的性能指标，即吞吐量、时延和处理消耗时间。吞吐量定义为面向文档数据模型的每秒处理增删改查操作的数量，如公式 3.2 所示。时延定义为面向文档数据模型的每个增删改查操作的平均时延，如公式 3.3 所示。消耗时间表示处理一组文档数据模型的增删改查操作所需时间，操作记录数是实验中的自变量。操作类型包括对文档数据模型中每条数据记录的插入、删除、更新和查询。为反映大规模数据处理，每次运行都执行 CURD 增删改查操作，数据记录数

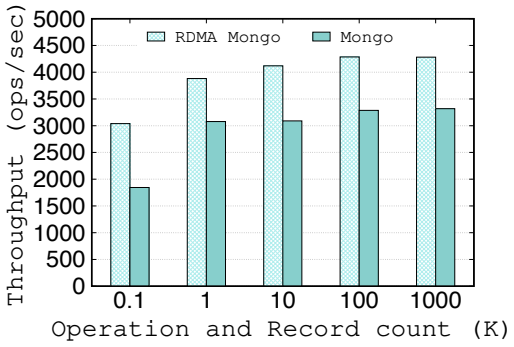
量从 0.1K 到 1000K 不等。为了综合比较，我们分别测量了单线程和多线程工作负载下文档数据模型端到端操作的消耗时间，每条统计测量的数据均通过执行 20 次相同的工作负载并将平均值作为最终结果，以确保文档模式模型端到端性能评测的准确性。在实验评估中，根据实际系统部署运行经验，公式 3.1 中的 k、S、B 和 T 的取值分别为 *0.7*、*50MB*、*100Gbps* 和 *32GB*。

$$P = \frac{op}{T} \tag{3.2}$$

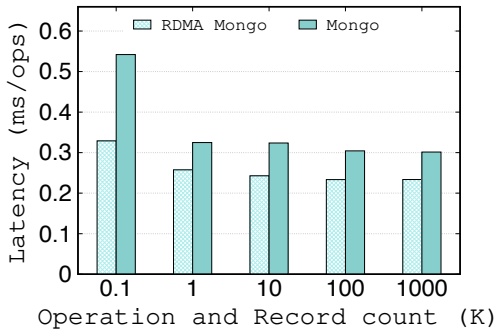
其中参数 P 指吞吐量，op 指数据操作计数，T 指所有数据操作的总时延。

$$L = \frac{T}{op} \tag{3.3}$$

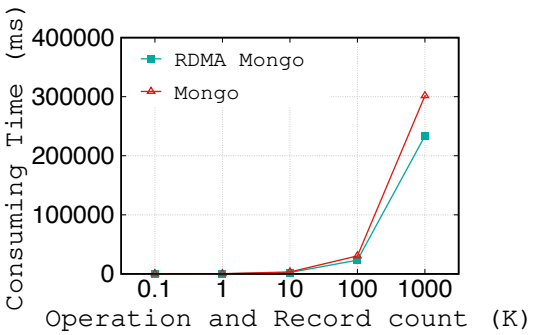
其中 L 是指每个增删改查操作的平均时延，op 是指数据操作计数，T 是指总时延。



(a) Insert 吞吐



(b) Insert 时延



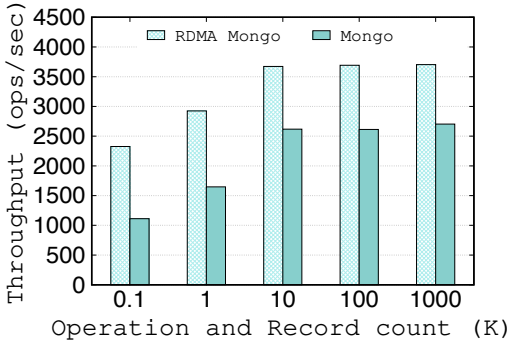
(c) Insert 总耗时

图 3-11　RMongo 与 MongoDB 的 Insert 操作性能对比

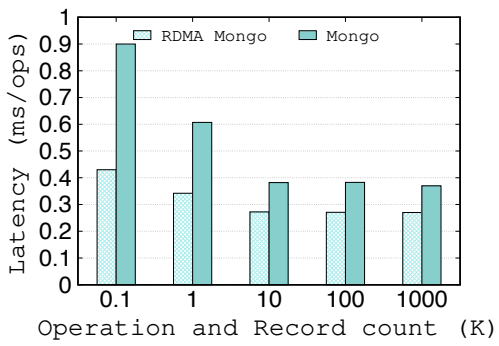
3.5.2　文档数据模型增删改查操作

在本节中，我们使用单线程增删改查操作评估 RDMA_Mongo 和传统 TCP MongoDB 的性能。在每次运行中，增删改查操作随着数据记录数的增加（0.01K-1000K）而执行，以比较与 RDMA_Mongo 和传统 TCP MongoDB 的性能。收集所有性能值后，自变量（操作和记

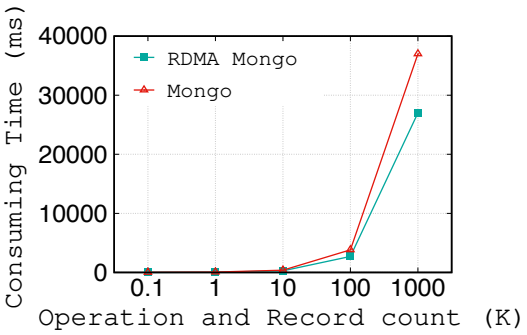
录计数）和性能指标（吞吐量、时延、消耗时间）之间的关系如图 3-11、3-12、3-13、3-14 所示。



(a) Delete 吞吐



(b) Delete 时延



(c) Delete 总耗时

图 3-12　RMongo 与 MongoDB 的 Delete 操作性能对比

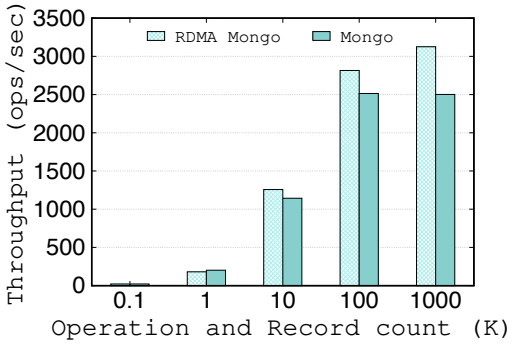
根据性能结果，我们发现 RDMA 增强范式可以显著地提升基于文档的 MongoDB 的 CURD 性能，插入、删除、更新和查询性能分别提高 30%、30%、17% 和 15%。

1. 数据插入性能

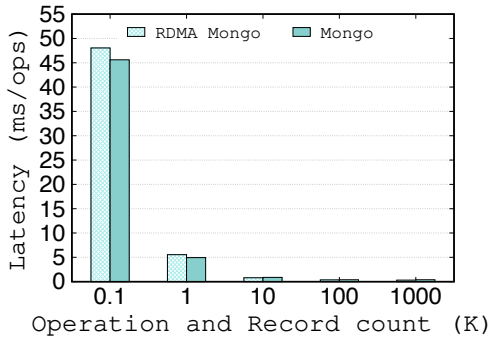
我们分别测量了在单个线程中插入 0.1K、1K、10K、100K 和 1000K 数据记录的总体消耗时间。每个操作插入一个数据记录。对于每个数据记录大小，我们测量消耗时间 20 次。之后，我们根据公式 3.2 和 3.3 计算吞吐量和时延。

图 3-11 展示了插入操作的吞吐量、时延和总体消耗时间。在吞吐量方面，我们可以看到 RMongo 比原生 MongoDB 具有更高的吞吐量，最大吞吐量差距为 *1194 ops/s*，最小吞吐量差距为 *804 ops/s*。具体来说，RMongo 的插入操作的平均吞吐量比原生 MongoDB 好 *29.72%*。同时，RMongo 的时延明显低于 TCP Mongo，最大时延差距为 *0.213 ms/ops*，最小时延差距为 *0.067 ms/ops*。就整体消耗时间而言，RMongo 与单边 RDMA Write 相比 TCP Mongo 实现的消耗时间要低得多。此外，0.1K 操作记录的消耗时间为 *21.3ms*，而 1000K 操作记录的消耗时间为 *67868.7ms*。我们发现，一旦操作记录数小于等于 10K，整体消耗时间的差距变得明显更大，这一优势来自于面向内存语义的单边 RDMA Write 原语直接基于 RDMA 内存区

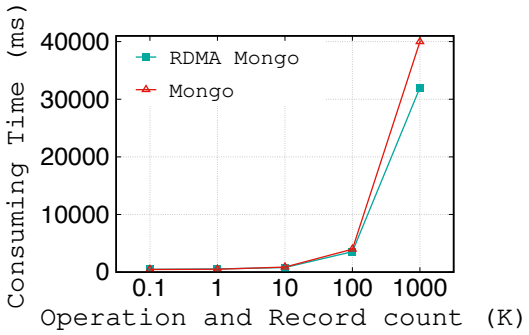
域重新组织文档数据模型，并结合零拷贝支持扩展了操作语义，从而减少了 RMongo 端到端插入操作路径上的内存拷贝和 CPU 上下文切换。



(a) Update 吞吐



(b) Update 时延



(c) Update 总耗时

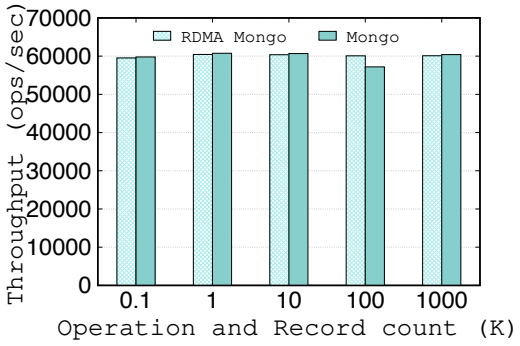
图 3-13　RMongo 与 MongoDB 的 Update 操作性能对比

2. 数据删除性能

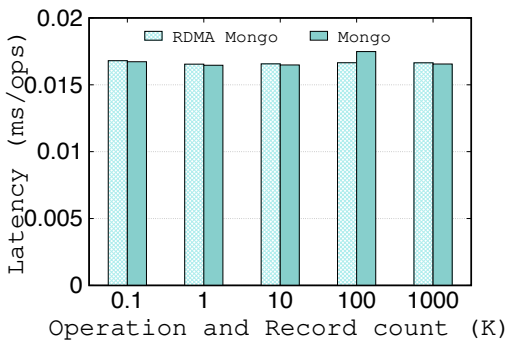
与数据插入操作的实验设置类似，我们分别测量了文档数据模型单线程删除操作负载分别在 0.01K、0.1K、1K、10K 和 100K 数据记录规模下的总体处理消耗时间。文档数据模型的每个删除操作会移除或软删除一个带有索引字段的数据记录。对于每种数据删除操作负载，我们均测量了 20 次的文档数据模型删除处理消耗时间。之后，我们根据公式 3.2 和 3.3 计算数据删除操作的吞吐量和时延性能。

文档数据模型删除操作的吞吐量、时延和总体处理消耗时间如图 3-12 所示。与传统 TCP MongoDB 相比，RMongo 实现了更高的吞吐量、更低的时延和整体消耗时间。对于吞吐量，RMongo 比 TCP MongoDB 高了 *37.03%* 至 *1.09* 倍，最大吞吐量差距为 *1276 ops/s*，最小吞吐量差距为 *702 ops/s*。在端到端时延方面，最大传输时延差距为 *0.265* 毫秒，而最小传输时延差距为 *0.068* 毫秒。这是因为，RMongo 利用了单边 RDMA Write 原语，实现了数据路径零拷贝，允许数据路径上绕过了服务端的 CPU。且随着文档数据模型的删除操作负载的增加，RMongo 与 TCP MongoDB 之间数据删除操作的整体处理消耗时间的差距明显更大，表

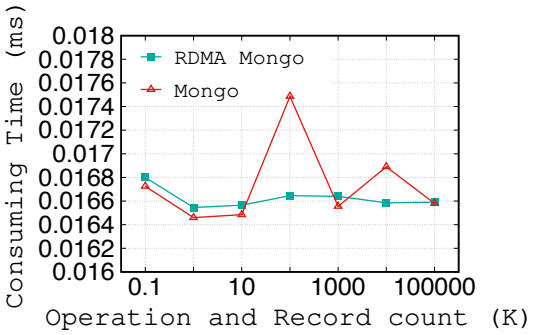
明 RMongo 内基于 RDMA 内存区域的文档数据模型设计在大规模的文档数据模型删除操作负载下性能优势更明显。



(a) Query 吞吐



(b) Query 时延

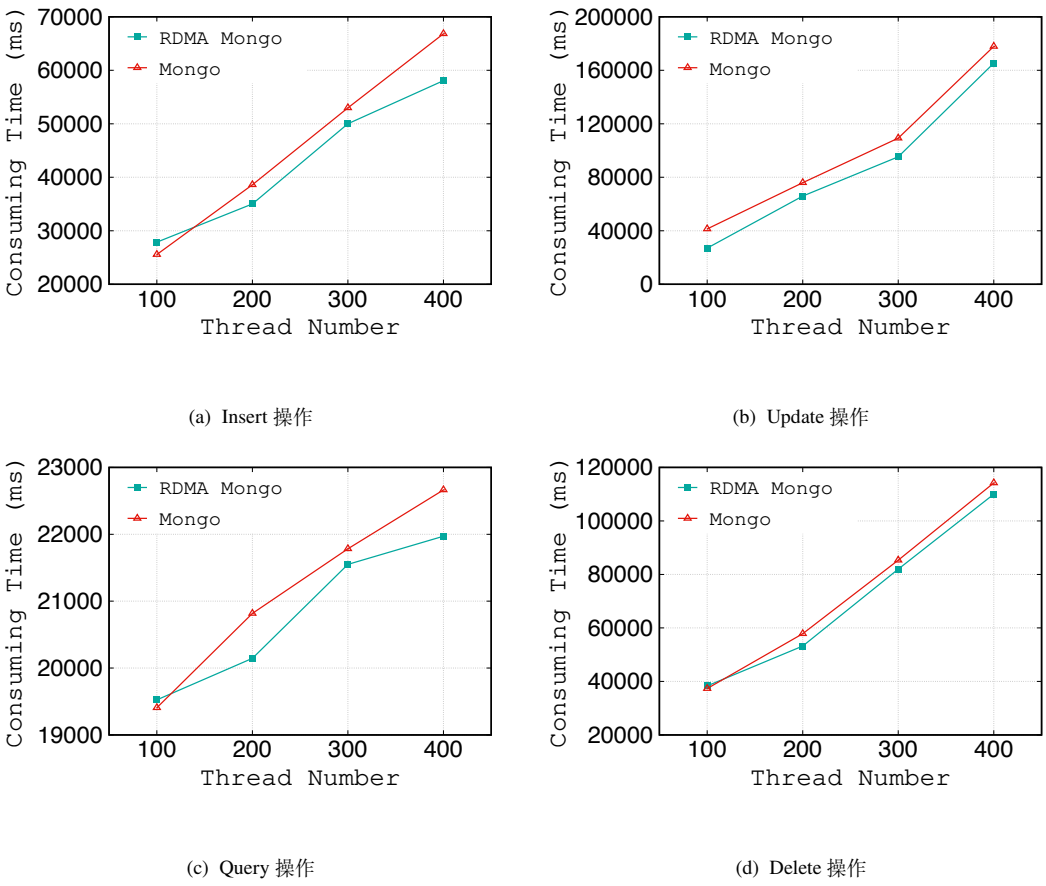


(c) Query 总耗时

图 3-14　RMongo 与 MongoDB 的 Query 操作性能对比

3. 数据更新性能

文档数据模型更新操作的实验设置类似于数据插入和删除操作。文档数据模型更新操作时，需要将新数据记录与旧记录进行比较，如果数据不一致，则执行更新操作。图 3-13 展示了 RMongo 和传统 TCP MongoDB 在不同大小的更新操作记录下的吞吐量、时延和总体消耗时间的比较。我们可以发现，对于小规模的文档数据模型更新操作负载（≤1000），RMongo 具有与传统 TCP MongoDB 相似的性能（吞吐量、时延、消耗时间），这是因为数据更新操作中新、旧数据记录之间存在数据一致性校验开销。然而，对于大规模的文档数据模型更新操作负载（≥1000），文档数据模型端到端更新操作性能由数据传输路径上的内存拷贝和 CPU 上下文切换开销主导，这种情况下 RMongo 实现了比基于 TCP 的 MongoDB 更高的更新操作吞吐量（高 10%~24.95%）。一方面，这得益于 RMongo 中基于 RDMA 内存区域的文档数据模型设计和 RDMA 单边 Write 原语扩展的数据操作语义，减少了数据路径的拷贝次数；通过预注册的 RDMA 内存池避免频繁的 RDMA 内存注册/反注册开销。另一方面，RMongo 采用 RDMA 的基于完成事件通道的异步通信模型和运行至终结模型（Run



(a) Insert 操作　　(b) Update 操作

(c) Query 操作　　(d) Delete 操作

图 3-15　在多线程并发的增删改查请求负载下，RMongo 性能明显优于完美的 TCP MongoDB

To Completition，RTC）相结合的方式重新设计了传输层的线程模型，避免了数据路径上的 CPU 线程上下文切换开销，充分利用了数据的缓存局部性。

4. 数据查询性能

针对单线程负载下的文档数据模型查询操作性能，我们分别测量了数据负载规模在 10、100、1K、10K、100K、1000K 和 10000K 数据记录下的总体查询处理消耗时间，每个查询操作针对多条数据记录进行检索。对于每种数据查询操作负载，我们均测量了 20 次的整体查询操作处理的消耗时间。然后，根据公式 3.2 和 3.3 计算文档数据模型查询操作的端到端吞吐量和时延。

图 3-14 展示了在递增的数据规模查询负载下文档数据模型查询操作的吞吐量、时延和处理消耗时间表现。我们可以看到，当查询的数据量较小（≤1000）时，TCP MongoDB 和 RMongo 的性能是相似的。然而，一旦查询的数据规模增大（≥1000），RMongo 相比于 TCP MongoDB 的查询性能收益是显著的。文档数据模型查询操作的数据规模越大，端到端查询操作的性能收益越明显。这是因为文档数据模型的单次查询操作的响应数据量通常大于数据插入、删除和更新操作，端到端查询操作过程中的数据拷贝次数是影响查询性能的主要因素，而 RMongo 直接在 RDMA 内存区域构建文档数据模型及相应数据结构，并通过单边 RDMA 原语扩展查询语义，绕过服务端 CPU 直接访问文档数据结构，相比于 TCP MongoDB

减少了内存拷贝次数。同时，RMongo 通过预注册的 RDMA 内存池和负载感知的 RDMA 缓冲区注册算法避免频繁的 RDMA 内存注册和维护产生的 CPU 开销，也有效减少了查询操作过程中单边 RDMA 原语传输引入的客户端与服务端之间往返同步查询的次数。而且随着单线程负载下数据规模的增加，传统 TCP MongoDB 的查询性能变得不稳定，而 RMongo 则有更加稳定的性能表现。这是因为 TCP MongoDB 在查询操作的数据路径处理至少需要内核线程与应用线程之间的上下文切换和多次拷贝，也引入了网卡到内核协议栈的硬件中断处理和内核协议栈到应用进程的软中断，且通信过程会阻塞查询逻辑的计算过程，这些均引入了较大的 CPU 开销，导致查询性能的较大波动；而 RMongo 充分利用基于 RDMA 完成事件通道的异步通信模型和 RTC 模型，既避免了数据接收处理路径上的线程上下文切换，又重叠了查询操作的通信和计算过程，从而带来更稳定的文档数据模型的端到端查询性能。

3.5.3　多线程并发负载性能评估

为了验证 RMongo 在大规模高并发请求负载下的文档数据模型的性能，多个文档数据模型 CURD 增删改查操作在 100 到 400 不等的并发线程数或客户端中同时执行。每个线程代表文档 NoSQL 系统客户端与服务端之间的一个已建立的连接，并执行 1000 个文档数据模型的增删改查操作。假设线程数为 N，每次运行同时启动 N 个线程，有 *N*1000* 条操作数据记录。在收集增删改查操作的整体处理耗时结果后，不同数据操作的线程数与处理时间开销之间的关系如图 3-15 所示。

根据图 3-15 中的性能结果，我们可以发现，在文档数据模型的多线程高并发增删改查工作负载情况下，RMongo 整体端到端处理消耗时间趋势线始终低于完美的传统 TCP MongoDB。具体来说，随着文档数据模型的增删改查并发请求规模的增长，相比于传统 TCP MongoDB，RMongo 总是能够实现更低的端到端处理时间开销。对于高并发的插入和查询性能，并发线程规模越大，RMongo 与 TCP MongoDB 之间的整体处理时间开销的差距就越大。当线程数为 400 时，*insert* 和 *query* 操作的消耗时间差距高达 15%。对于大规模并发数据请求和处理，文档数据模型端到端操作过程中频繁的内存拷贝和 CPU 上下文切换是主要开销。通过比较分析文档数据模型的端到端操作处理消耗时间，RMongo 在多线程并发工作负载下的性能收益与单线程情况下的性能效果类似，均显著优于完美的 TCP MongoDB。这是因为文档数据模型在高并发端到端增删改查请求负载下的性能受到多线程之间内存和计算资源的约束与竞争开销；对于多线程之间的内存开销，RMongo 利用基于 RDMA 内存区域的文档数据模型设计和操作语义扩展，实现了零拷贝支持，相比于传统 TCP MongoDB 至少减少了 2 次内存拷贝，显著增加了高并发负载下的可用内存资源量。对于多线程负载下的 CPU 计算开销与潜在 CPU 瓶颈，RMongo 通过单边 RDMA 原语扩展文档数据模型的增删改查操作语义，绕过服务端 CPU 直接访问文档数据结构，显著地缓解了文档 NoSQL 系统服务端的潜在 CPU 瓶颈；RMongo 采用基于 RDMA 完成事件通道的异步通信模型来重叠通信和计算过程，并结合完成事件的批量处理优化，进一步降低了文档数据模型在高并发负载下的 CPU 开销。此外，考虑到大规模并发 RDMA 可靠连接会竞争 RDMA 网卡硬件上的缓存资源，RMongo 采用了 RDMA 共享接收队列技术来缓解 RDMA 网卡缓存资源的争抢开销，从而提高了端到端数据操作性能的扩展性。同时，文档数据模型的高并发操作请求会经由 RMongo 的路由组件分发到多个副本分片节点，而基于 RDMA 单边原语和异步事件模

型的多分片节点之间操作日志同步协议，增强了文档数据的副本同步效率，有效地提高了高并发场景下文档数据模型的可用性。

3.6　本章小结

受 RDMA 增强范式的启发，我们提出了 RMongo，一个 RDMA 增强的高吞吐和内存资源高效的文档数据模型及文档 NoSQL 系统。它可以有效地利用无损 RDMA 网络中内核旁路和零拷贝特性来缓解传统的网络传输挑战：频繁的 CPU 参与和高并发 MongoDB 客户端的等待时间长。在本文中，我们特别分析了 MongoDB 网络传输层。此外，我们阐述了丰富的设计空间和三种设计选择之间的权衡。RMongo 采用新颖的 RDMA 上下文检测算法和负载感知缓冲区注册算法，与传统网络堆栈共存，并以负载感知的方式注册 RDMA 通信区域。为了加速主从节点之间的操作日志同步，利用 RDMA 立即数据写入发送控制消息或操作日志，而 RDMA 完成事件通道用于异步接收消息或操作日志。以 MongoDB 客户端和 mongos 之间的 CURD 请求为例，RMongo 使用更高效的单向 RDMA 读写原语和可靠连接传输模式实现了一种协调机制。在基线操作和高并发连接数上的实验结果表明，RMongo 的性能得到了很大的提升。展望未来，我们致力于通过研究基于 RDMA 的分片策略、事务机制和基于非易失性内存（Non-Volatile Memory，NVM）的存储，使基于 RDMA 的更高性能文档 NoSQL 系统成为可能。

第 4 章　BoR: RDMA 驱动的分布式共识协议及许可区块链系统

本章的主题是基于 RDMA 增强时延/CPU 敏感型系统，以基于分布式共识协议的许可区块链系统作为典型案例展开研究，在本文中所处位置如图 4-1 所示。本章从网络和时延/CPU 敏感系统协同设计的角度，将提出一种 RDMA 驱动的高性能、CPU 高效和可扩展的分布式共识协议，并在此基础上实现了对应的许可区块链系统 BoR。

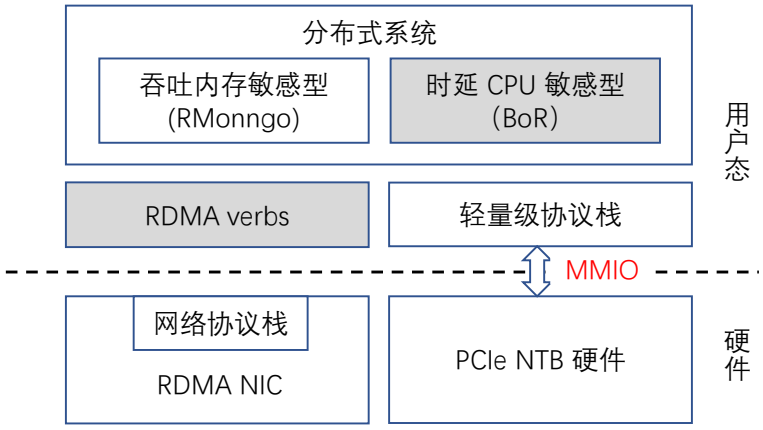


图 4-1　本章主题：基于 RDMA 增强吞吐/内存敏感型系统，用灰色背景的方框标记

4.1　引言

许多时延/CPU 敏感性系统已经在云供应商的数据中心内广泛地部署，典型系统如分布式共识协议驱动的区块链即服务（Blockchain as A Service，BaaS），是一种新兴的面向多租户的一致性数据服务模型，在易部署性、透明度、效率和成本之间提供了有效的权衡。一方面，BaaS 云提供商专注于为租户维护分布式共识协议及区块链的重要组件和基础设施。BaaS 支持的后端活动包括共识网络的部署、合适的资源分配、安全特性和租户的托管要求。另一方面，租户可以通过使用 BaaS 云服务专注于基于分布式共识及区块链的应用程序的构建，而无需担心与性能相关的问题，从而兼顾了灵活性和性能。许多云提供商，包括 AWS Cloud[102]、Microsoft Azure[103]、Oracle 和 IBM 云[104]，已经将基于云的 BaaS 大规模部署到数据中心。

随着数据规模的递增和数据一致性服务的高并发请求，时延/CPU 敏感的 BaaS 需要低时延、高吞吐的分布式共识协议驱动。为此，许多研究工作致力于 BaaS 的性能分析[47–49, 109]和分布式共识算法[50–52, 107, 108, 110, 111]。早期的 PoW 驱动的区块链如比特币[112] 和以太坊[106]需要大量的算力进行哈希计算，这导致交易吞吐量非常低，CPU 消耗高，出块间隔大。由

表 4-1　4 个不同的分布式共识协议及区块链系统的性能对比

系统	区块间隔	分布式共识	事务吞吐
Bitcoin	10 minutes	PoW[105]	about 7
Ethereum	10 to 20 seconds	PoS[106]	< 100
Fabric	3 to 6 seconds	PBFT[107]	> 1000
EoS	0.5 seconds	DPoS[108]	million

于在分布式共识算法[50–52, 107, 108, 110, 111]上的巨大努力和优化，区块生成间隔已经从几分钟减少至几秒钟，甚至毫秒级，如表格 4-1 所示。Hyperledger Fabric[107] 采用了 PBFT[110] 共识，对应的区块生成间隔为 3.6 秒，而 DPoS[111] 已用于 EoS 系统[108]，对应的区块生成间隔仅为 0.5s。这种高效的分布式共识算法消除了无价值的计算资源开销，并导致更高的区块事务吞吐量。因此，BaaS 的性能瓶颈已经潜在地转移到区块数据的传输开销上，包括新节点的区块广播时延和新节点引导协议处理时延。

表 4-2　不同分布式共识协议驱动的区块链系统的数据规模与初始同步时间

系统	数据规模	增长率	初始同步时间
Bitcoin	170GB	50GB per year	2-3 days
Ethereum	430GB	270GB per year*	> 2 days
EoS	330GB	12GB per month**	1-2 days

* From 2016 to 2017

** The first month of EoS

此外，对于更大规模的 BaaS 云服务，被延迟的区块传输将导致更多的分叉（如图 4-2 所示），具有更高的被攻击概率。此外，随着区块链网络的快速发展（如表格 4-2 所示），引导新节点的时延线性增加，这严重降低了区块链系统的可扩展性。尽管进行了许多研究，但传统的基于 TCP 的区块链在交易吞吐量、区块广播时延、CPU 开销和新节点的引导时间方面面临着许多挑战[113]。因此，在面向时延/CPU 敏感的分布式共识协议及区块链中，如何减少无意义的 CPU 参与，最小化区块广播时延和新节点的引导处理时延，是具有挑战性的。

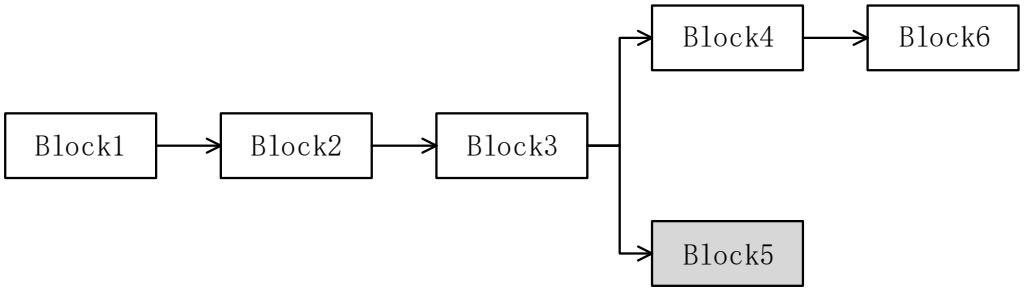


图 4-2　区块链的分叉 *fork* 行为

而新型网络通过协议栈硬件卸载提供超低时延、高带宽、CPU 绕过等高级硬件功能，如商用高速 RDMA 网络[29]，为时延/CPU 敏感的应用程序的整体性能增强提供了新机遇[45, 80]。

此外，融合以太网上的 RDMA 硬件实现 RoCE 已广泛部署于许多生产级的数据中心，以加速大规模数据的高性能传输和处理[10, 35, 48]。数据中心中的许多应用程序利用 RDMA 原语（例如 RDMA Write/Read/Send）[54] 来覆盖底层通信机制以获得更高的吞吐量、更低的时延和更少的 CPU 参与，这被称为 RDMA 增强范式。例如，Wukong[45] 提出了 RDMA 驱动的基于图的 RDF 存储系统，用于大规模数据集上的低时延和高并发查询服务。RDMA_Mongo[114] 采用单边 RDMA 原语来重新设计 oplogs 同步，以便在基于文档的 NoSQL 中更快地处理大规模数据。Octopus[58] 利用 RDMA 和 NVM 实现了低时延的面向持久化内存的高性能文件系统。其他支持 RDMA 的系统专注于键值存储[35, 35, 37, 40, 55]、分布式文件系统[80, 81] 等。因此，我们能否基于内核旁路的 RDMA 网络，增强时延/CPU 敏感的分布式共识协议，以提高云数据中心 BaaS 的服务质量?

受 RDMA 增强范式的启发，我们提出了一种面向商用 RDMA 网络的高性能、CPU 高效、可扩展的分布式共识协议及对应的许可区块链，称为 **BoR**。首先，当本地节点与其对等节点建立连接时，我们设计了一种 RDMA 上下文检测方法。如果本地和远程节点均在互连的 RDMA 网络中，该方法通过 TCP 连接交换同步 RDMA 链路状态和元数据，节点之间新建连的 RDMA 会话将被推送到本地传输通道列表中。否则，BoR 将回退到传统 TCP/IP 链路执行分布式共识协议。其次，本地节点向所有 TCP 连接和 RDMA 通道发送握手消息以进行状态同步。并将收到显式的通知消息，包括远程对等节点上的区块和事务的状态。本地节点根据此类通知消息确定是否需要进行区块数据同步操作。传输数据可分为确认消息、控制消息、事务和区块。BoR 利用带有即时数据的 RDMA Write 原语扩展了分布式共识协议，然后采用 RDMA 完成事件通道与异步通信模型的协同，来实现共识消息的高性能收发。最后，我们基于混合 RDMA 原语和共享接收队列重新设计了区块/事务数据的同步协议。考虑到注册和注销 RDMA 内存区域的 CPU 开销，我们为每个 RDMA 通道预注册足够的内存区域，并设计了面向 RDMA 内存区域管理的内存池。每当需要发送或接收区块数据共识的请求/响应消息时，RDMA 通道只需要从 RDMA 内存池中申请一个内存区域分片，并获取相应的内存地址和远程密钥。

我们的主要贡献总结如下：

- 我们对当前时延/CPU 敏感的分布式共识算法进行了详细分析，然后确定了整体设计目标和挑战，并提出了基于 RDMA 的分布式共识协议及系统的整体架构，称为 BoR。
- 我们提出了一个支持 RDMA 的同步管理器，用于管理 RDMA 会话通道、内存区域、RDMA 适配器和区块同步状态。
- 我们基于带有即时数据的 RDMA 原语，重新设计了一个新的节点引导和区块/事务广播协议，并基于完成事件通道的异步通信模型来保证低时延、高吞吐量和更少的 CPU 参与。
- 我们基于 EoS 系统构建了 BoR 的系统原型，并利用不同数据规模的工作负载评估分析了 BoR 与原生 EoS 系统的性能差异。与基于传统 TCP 网络的原生 EoS 系统相比，当新节点加入区块链网络时，BoR 可降低初始区块同步的时延达 20.2%，降低了 26.4 至 33.9% 的 CPU 资源开销。

本章的其余部分组织结构如下。第 4.2 节介绍了时延/CPU 敏感的分布式共识协议、RDMA 增强系统的背景知识，然后解释了基于内核旁路的 RDMA 网络增强分布式共识协议及区块链的动机。第 4.3 节概述了基于 RDMA 网络的 BoR 系统的整体架构，然后讨论了面向时延/CPU 敏感的分布式共识协议的设计目标和挑战。在第 4.4节中，我们描述了 BoR 系统的关键设计思想，然后介绍了 RDMA 驱动的初始同步和区块广播机制。在第 4.5 节中，我们评估 BoR 系统，并将其与原始 EoS 区块链进行性能比较和分析。第 4.6 节介绍了商用 RDMA 网络与时延/CPU 敏感的分布式共识系统的协同设计研究的结论和未来工作。

4.2　背景与动机

4.2.1　分布式共识驱动的区块链系统

近年来，随着比特币[112]、以太坊[106] 等新兴加密货币的蓬勃发展，底层支撑的分布式共识算法和区块链受到了业界的广泛部署和研究。区块链是一个分散的分布式账本或数据库[115]，它由基于区块单元的链式数据结构[113] 组成，其数据结构如图 4-3 所示。它旨在实现无需第三方信任保证的透明多方事务服务。利用拜占庭容错共识协议[116]（如可证明投票机制[117]）来保证区块链节点之间的一致性，每个加入共识网络的新节点均维护了一个全量的区块数据账本；利用可扩展的点到点网络[118] 来驱动去中心化的分布式共识算法。数字签名[119] 和 Merkle 树[120] 被用于防止区块链上的事务记录被恶意篡改。智能合约提供了区块链平台的可编程性，区块链中维护的每一条数据记录都只能通过运行智能合约来访问和追踪[121]。

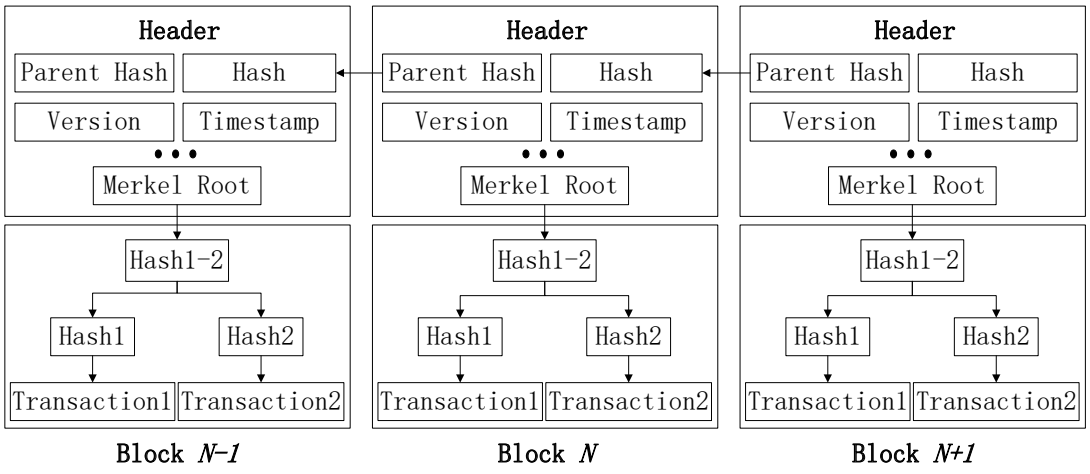


图 4-3　面向分布式共识算法的区块数据结构

下面的三个关键概念是分布式共识算法驱动的区块链通用架构的三个关键组件，如图 4-4 所示：

事务（Transaction）。作为资产或价值的载体，一笔事务可以广播到区块链网络并收集到区块中。通常，先前的事务输出被事务引用为新的事务输入。只有当足够数量的可信对等节点遵循分布式共识协议，对新生成的区块/事务达成共识时，区块中的事务才能被转化为不可

逆转的事务。区块事务在区块链中未加密，从而可以浏览所有已聚合到区块中的事务。

区块（block）。一个区块可以被视为一批事务的集合，每个事务条目都持久化地存储在对应的区块文件中。随着去中心化的分布式共识算法的处理，所有区块实体都被组织成一个链式的数据结构，如图 2 所示，区块头和区块体组成了一个区块实体。其中，区块头包含元数据，包括父哈希、本地哈希、时间戳和默克尔根，而区块体由组织成默克尔树的事务组成。矿工处理的新事务以新区块的形式聚合到默克尔树中，这些区块附加到区块链的末尾。

共识（consensus）。区块链的运行原理是通过时延/CPU 敏感的分布式共识算法来保证链上数据的完整性和强一致性，共识的关键目标是防止不必要的分叉或恶意篡改。

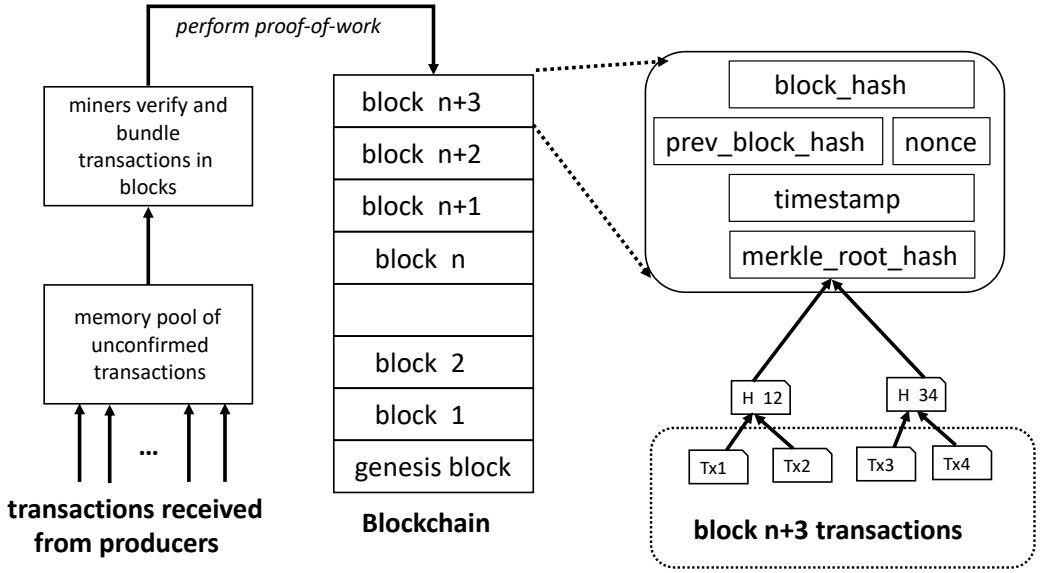


图 4-4　区块链平台的通用架构

基本上，区块链可以分为公共链、私有链或许可链。在公共区块链（如比特币[112] 和以太坊[106]）中，任何人都可以被允许参与、离开、审计、访问和编写正在进行的活动。这有助于维护自治功能。与公共区块链相比，私有或许可的区块链如 Hyperledger Fabric[122] 和 EoS[108] 通常部署在数据中心，旨在构建一个参与方互相感知、面向联盟或组织机构的高可信度平台。本文重点关于在时延/CPU 敏感的分布式共识协议及对应的区块链系统的设计与实现。

然而，随着历史区块数据集规模的不断扩大和区块链架构的快速演进以及更快的共识，区块链中基于传统 TCP/IP 点到点通信很难满足高并发共识负载下低时延、高吞吐的要求。传统区块链的底层网络协议栈是一个潜在的瓶颈，特别是对于新节点加入区块链网络时的初始区块同步。

区块链即服务。区块链即服务（BaaS）是一种基于云的解决方案，它提供分布式区块链运行时以满足租户所需的高灵活性和可用性。云提供商专注于管理与区块链相关的后端服务、组件和资源分配，而租户直接专注于开发功能、智能合约和区块链应用程序[109, 115, 119, 123, 124]。因此，近些年 BaaS 模型的设计与实现吸引了许多研究人员和机构的研究兴趣。例如，Blockstack[115] 是一个基于比特币的命名和存储系统，其中最少的元数据存储在比特币中。先前的工作[124] 采用基于区块链的云加密搜索方案来抵御恶意攻击。

然而，随着基于智能合约的应用程序数量的和部署规模的不断扩大，云数据中心 BaaS 服务的高请求率产生了对高性能分布式区块链系统的强烈需求。虽然之前的系统[50, 51, 109] 已经提出了一些有效的优化策略和算法来解决性能差距，但它们在利用丰富的现代硬件（如数据中心的 RDMA 设备）来实现更高质量的 BaaS 方面存在不足。

EoS 和 Graphene 技术。 *EOS* 是 Blockone 开发的新兴区块链平台。EoS 的目标是实现一个面向区块链架构的操作系统，通过智能合约编程抽象的方式为上层应用程序提供数据完整性、防篡改、一致性共识、机密性计算等功能[108]。由于传统区块链应用存在高时延和低吞吐量的限制，EoS 采用高效的并行链和 DPoS 共识来推动更快的区块链。与 PoW 和 PoS 共识机制不同，DPoS 节点需要参与见证选举。只有赢得选举的节点（至少 21 个节点）才有权生成区块。此外，另外 100 个候选节点被视为见证候选。一旦 21 个见证节点出现问题，它们就被视为替代品。EoS 的当前出块间隔约为 0.5 秒。

EoS 中包含三个主要组件：*nodeos*、*cleos* 和 *keosd*。Nodeos 充当服务器节点组件。Cleos 是一个命令行接口，负责与区块链交互，管理钱包和账户，调用区块链上的智能合约。Keosd 被用来管理 EOSIO 中的钱包。

石墨烯技术（*Graphene Technology*）是 BitShares 团队开发的一个区块链工具包，可以实现更高的并发水平。它是一个开源的区块链底层库，基于 DPoS 共识，并已在 EoS 中使用。借助石墨烯技术，EoS 可以通过平行链达到每秒数百万次事务（Transaction Per Second，TPS）。EoS 采用本地并行链技术，从而实现了面向分布式共识算法的毫秒级区块共识速度。由于 EoS 在共识和 TPS 方面的高性能，本文采用 EoS 作为构建 BoR 系统原型的基础。

4.2.2　RDMA 特性与分布式共识

远程直接内存访问（RDMA）是一种完全绕过内核的高速互连技术，通过将整个 TCP/IP 协议栈卸载到专用 RDMA NIC（RNIC）中，具有内存语义。如图 4-5 所示，RDMA 利用无损网络来实现高带宽，同时采用零拷贝技术来减少系统中断和内存拷贝的数量，从而实现超低时延。RDMA 提供了两种面向消息的原语：Send/Recv Verb 等双面消息传递接口和 Write、Read 和原子操作（Compare-and-Swap 和 Fetch-and-Add）等单边原语。与单边 RDMA 原语相比，双边原语操作仍然需要 CPU 参与远程主机和用户空间的内存拷贝，这引入了不可避免的开销[44, 125]。为了最大限度地减少初始区块同步期间不必要的开销，我们利用内置即时数据的单边 RDMA Write 原语处理带宽敏感的数据消息（例如区块和事务）。为了保证 BoR 中区块/事务传输的高可靠性，充分发挥 RDMA 内存语义的高性能，本文关注于 RDMA 的可靠连接模式。

正如之前的工作[89] 所得到的经验性结论，不同 RDMA 原语（两边或单边原语）的适当组合可以提高 RDMA 增强系统的性能。因此，利用双边 RDMA Send/Recv 原语在 BoR 节点之间交换注册的消息缓冲区，而在 BoR 中的区块广播和同步期间，我们利用内置即时数据的单边 RDMA Write 原语进行面向分布式共识协议的控制和数据消息的传输。

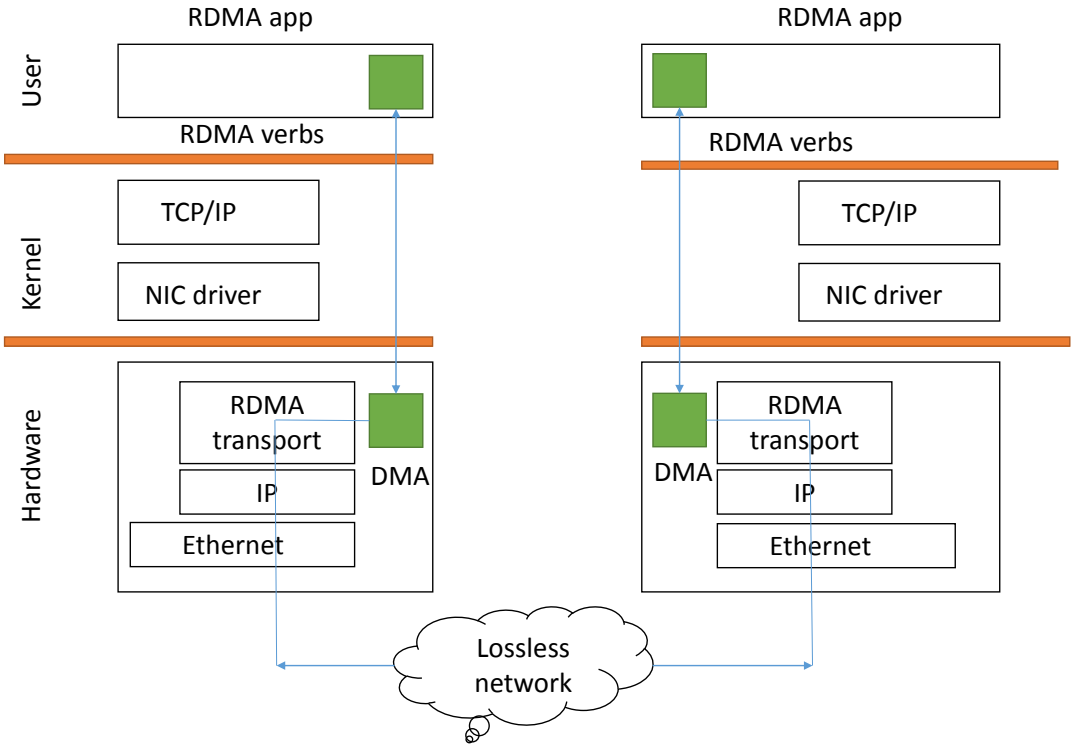


图 4-5　RDMA 驱动的通信特性

4.2.3　动机

区块链即服务 (BaaS) 已广泛部署到许多云数据中心[102–104]，并向租户和开发人员公开了与区块链相关的编程接口。大量基于智能合约的应用程序采用基于云的 BaaS 作为其底层的区块链运行时。随着 BaaS 的大受欢迎以及 BaaS 上智能合约的数量不断增加，BaaS 不断增长的请求率需要更快、更高并发的许可区块链。然而，由于传统基于 TCP 的区块链的冗余内存拷贝、频繁的 CPU 上下文切换和传输层中断带来的不可避免的开销，现有的解决方案[50, 108, 109, 122] 无法实现所需的 BaaS 质量，如高吞吐量和低时延。另一方面，更快更高效的共识算法使得区块同步和广播成为潜在的瓶颈。同时，由于其超低时延和高吞吐量以及远程 CPU/OS 绕过，数据中心的现代 RDMA 硬件已被广泛用于增强数据密集型的应用系统，例如 NoSQL 系统[35, 37, 40, 45, 55, 114] 和分布式文件系统[80, 81]。RDMA 支持的内核绕过功能减少了 CPU 上下文切换和系统中断的次数，而网络协议栈硬件卸载和内存语义的零拷贝技术显著地减少了有效载荷拷贝的次数，以实现低时延、高消息速率和低内存总线争用。受此启发，我们尝试利用混合 RDMA 原语（RDMA Write 和 Send/Recv）来加速 BaaS 的初始区块同步和区块/事务传播。据我们所知，BoR 是率先尝试利用 RDMA 技术来提升基于云的 BaaS 的高性能许可区块链①。

4.3　整体架构概览

① 在 2019 年 10 月 BoR 工作录用时，据我们调研所知，尚未存在基于 RDMA 的分布式共识协议和区块链系统的相关研究工作。

4.3.1　设计目标

更高的区块同步效率。刚加入分布式共识网络的新节点会首先同步区块链中的所有历史区块。完成区块同步需要 1 到 3 天的时间（如表格 4-2 所示），区块同步效率很低。更糟糕的是，在新节点的初始区块同步过程中，该区块链中的生产者会继续生成新块，从而导致新节点同步新生成区块的时延开销更大。近年来，随着快速高效的新型分布式共识算法的研究演进，共识过程中区块间隔被显著地降低了。如表格 4-1 所示，EoS 中最短的区块间隔为 0.5 秒。因此，如果同步速度低于生成速度，例如每个区块同步 1 秒，生成每个区块 0.5 秒，则新节点将始终处于同步最新区块的状态，很难完成区块数据的同步。由于网络传输会极大地影响同步速度，因此我们的第一个设计目标是提出一种使用内核绕过 RDMA 技术同步区块的有效方法。

更低的共识广播时延。当生产者生成一个新区块时，它会立即将其广播给所有其他对等方以进行基于共识的确认。其他收到区块的节点将通过共识协议（如 PoW 或 DPoS）对其进行验证，然后将验证结果（有效或无效区块）广播到整个区块链网络。由于区块生成间隔较短（0.5 秒），当前的 EoS 系统需要更快速高效的分布式一致性共识协议。

这意味着，如果一个新区块由于网络时延或其他问题长时间没有被其他节点验证，下一个新区块很大概率在前面的区块被共识确认之前产生。大量累积的未确认区块可能会显著降低 BaaS 云中区块链系统的整体性能。如何降低区块广播的传输时延，是提升分布式共识协议的性能的一个重要挑战。因此，BoR 的第二个设计目标是在 RoCE 网络中引入基于 RDMA 的广播方法来加速区块广播。此外，我们提供了一种具有较低广播时延的轻量级 RDMA 增强共识机制。

更高的端到端吞吐和更低的 CPU 开销。除了区块链业务层面的优化，从系统角度优化区块链也很关键。由于基于共识的哈希计算对计算能力的需求以及初始块同步引入的高 CPU 利用率，现有的区块链系统资源分配不足且效率低下。传统 TCP 的区块链系统的传输层由于不可避免的内存拷贝、频繁的 CPU 上下文切换以及昂贵的操作系统调用开销而占用 CPU 资源。为了最大限度地减少上述开销并降低 CPU 资源开销，BoR 利用 RDMA 零拷贝特性和混合 RDMA 原语（即 RDMA Send/Recv 和带有即时数据的 RDMA Write 原语）来减少系统资源的消耗（即降低 CPU 开销），从而提高区块链网络的吞吐量。此外，区块/事务转移中节省的资源可以被其他区块链操作（如基于共识协议的哈希计算）利用。因此，我们重新设计了一个在 RDMA 网络上更有效的区块同步机制。

4.3.2　BoR 架构

BoR 是具有分布式数据库结构的 RDMA 增强型分布式账本。它由三个主要组件组成：*nodeos*、*keosd* 和 *cleos*。Nodeos 是运行在 BoR 服务器上的一个核心进程，它由链管理层、区块存储和传输层组成。它承载着与区块链网络交互的基本功能，如配置系统参数、管理区块链状态、处理事务/区块请求、管理本地区块记录等。Keosd 是一个钱包管理客户端，主要用于管理公钥/私钥和钱包信息。Cleos 是一个交互式命令行工具，它连接用户和 keosd/nodeos。本文重点介绍 nodeos 组件。我们为 nodeos 实现了一个支持 RDMA 的传输层，它可以以 RDMA 友好的方式与链管理器层和区块存储协同，如图 4-6 所示。

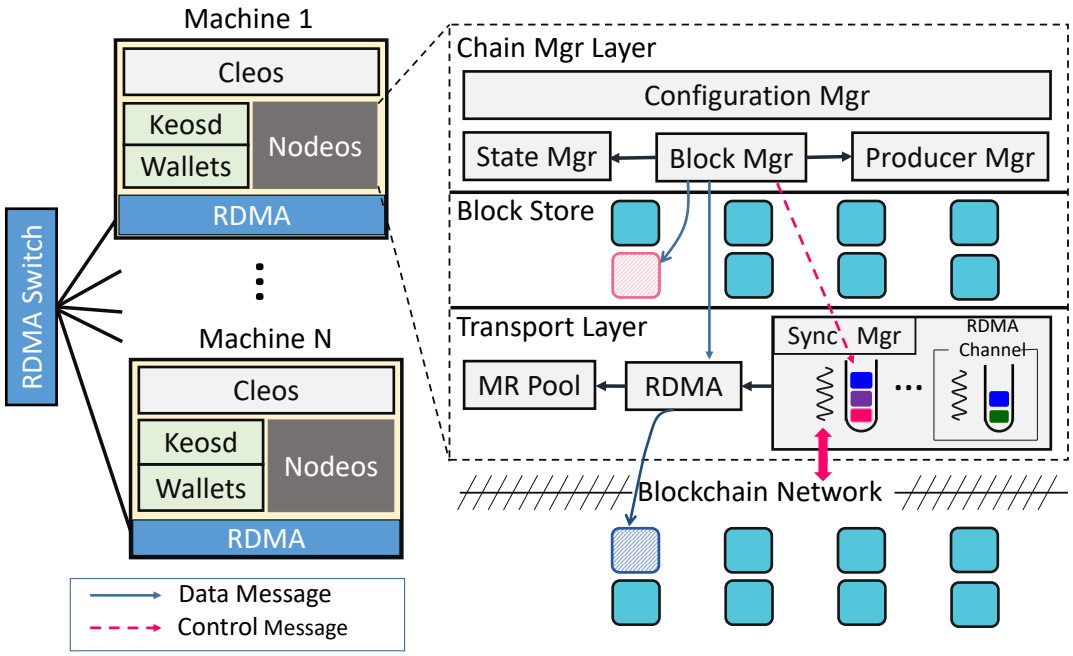


图 4-6　BoR 系统架构概览

区块存储 (Blocks store) 是一个存储层，用于在本地机器中存储区块/事务。它可以向链管理器层的区块管理器提供对区块/事务的查询、更新和删除操作。

链管理层 (Chain manager layer) 称为链管理器层，由配置管理器（*Configuration Mgr*）、状态管理器（*State Mgr*）、区块管理器（*Block Mgr*）和生产者管理器（*Producer Mgr*）组成。配置管理器主要负责读取 BoR 配置文件（如 *config.ini* 和 *genesis.json*），并向状态管理器、区块管理器和生产者管理器提供相应的配置参数。状态管理器用于管理本地块/事务的状态。例如，被共识协议确认的区块/事务被视为不可逆的区块/事务，而未被确认的区块/事务被视为可逆的区块或事务。利用生产者管理器来生产/签署新区块并验证区块、签名或事务的合法性。

区块管理器可以从/向传输层接收或发送控制/数据消息。具体来说，当区块管理器收到来自 cleos 的请求时，它会生成相应的响应控制消息，并将其推送到传输层 RDMA 通道的发送队列中，如图 4-6 所示。当从传输层接收到控制消息时，区块管理器将采取与不同消息类型（如握手和请求消息）相对应的动作。这些操作包括从状态管理器请求区块链状态或从生产者管理器请求签名。例如，当收到同步请求消息时，区块管理器解析有效数据载荷中的块头编号或事务 ID，并在区块存储中查询相关的区块/事务。然后将结果按照分布式共识协议封装为响应数据消息，并通过单边 RDMA 原语传输到目标节点。

传输层 (Transport layer) 由三个主要组件组成：内存区域（MR）池、RDMA 传输组件和同步管理器。利用预先分配的 MR 池来减轻运行时 MR 注册和注销的开销。支持单边原语的 RDMA 传输组件为带宽敏感的数据消息（如区块/事务）提供高效传输。同步管理器管理一组 RDMA 通道，每个 RDMA 通道代表本地节点和一个对端节点之间的会话或连接。RDMA 通道使用基于事件通道的工作线程异步接收控制/数据消息并回调相应的处理程序。此外，同步管理器还可以从链管理器层的区块管理器接收消息并将它们发送到区块链网络。

4.3.3　设计挑战

在许可区块链共识协议中采用多种消息类型还是统一数据类型？在 EoS 中，一共有 9 种面向分布式共识协议的关键消息类型。常用的消息有握手消息 *handshake_message*、通知消息 *notice_message*、请求消息 *request_message*、以及区块同步请求消息 *sync_request_message*。这四种类型的消息充当不同对等节点之间的控制消息，涉及不同的字段以实现不同的特性。还有另外两种类型的消息，*signed_block* 和 *pack_transaction*，它们传输由生产节点生成的区块或事务。接收端通过重载函数 *handle_message()* 处理消息。挑战在于如何有效地管理这些不同的消息类型。BoR 利用带有即时数据的单边 RDMA Write 原语设计了时延敏感的控制消息传输以及对带宽敏感的数据消息传输协议。但是，基于单边 RDMA 原语的数据传输需要预先注册的 RDMA 内存区域（Memory Region,MR）来发送和接收消息，这导致为针对 RDMA 内存语义传输的未知消息的 RDMA 缓冲区大小分配存在很大挑战。当多种类型的消息具有不同的消息大小时，接收端无法自适应地预测即将到来的消息类型。尽管消息类型是由发送方在消息结构中定义然后通知接收方的，但引入的开销会降低整体性能。因此，我们使用统一的消息定义，包括 BoR 中所有类型的消息，以简化内存缓冲区的分配、内存池的管理以及消息的序列化和反序列化。只需要一个面向 RDMA 内存语义的消息数据结构抽象，就可以表示面向分布式共识协议的多种消息类型。

BoR 系统中 RDMA 通道的管理。另一个挑战是如何有效地管理 RDMA 通道。如第 2 节所述，RDMA 驱动范式需要具有 RDMA 功能的专用 NIC，并且只有具有互连 RNIC 的机器才能享受通过强大的 RDMA 原语进行通信的好处。但是，我们无法确保参与分布式共识协议的每个共识节点均支持商用 RDMA 网络链路。如果没有部署 RDMA 网络硬件，机器就无法使用 RDMA 通道与其他节点通信。为了解决这个问题，我们提出了一种混合解决方案：同时保留 TCP 通道和 RDMA 通道。采用一种新颖的 RDMA NIC 检测算法来确定不同机器之间通过 RDMA 的通信是否可用。如果可用，两个对等节点将利用支持 RDMA 的传输进行区块/事务同步。否则，BoR 采用基于传统 TCP/IP 的传输层来驱动时延/CPU 敏感的分布式共识协议。在 BoR 系统启动期间，RDMA 上下文检测算法依赖于传统 TCP 链路来同步校验多个计算节点之间的 RDMA 链路状态。

RDMA 驱动的区块链上异步请求处理。 BoR 客户端的请求到达通常需要 BoR 服务器产生相应的响应消息。在高并发的情况下，应该异步处理客户端请求消息，以减少阻塞时间。在 BoR 执行过程中，异步处理可以像占用线程一样释放资源避免阻塞，等待响应消息产生，然后重新获取一个可用线程进行请求处理。挑战在于如何使用 RDMA 进行异步处理。RDMA 尚未提供已包装的异步处理程序。因此，我们创建了一个监听线程，它利用 RDMA 完成事件通道异步接收缓存在 RDMA 完成队列（CQ）中的消息。然后，在侦听线程中的事件通道持续侦听的同时，附加到完成队列元素（CQE）的消息被解析并分发到相应的消息处理程序中。在消息处理程序中，CQE 中的操作码字段用于从不同的原始操作中识别 *IBV_WC_RECV_RDMA_WITH_IMM* 类型，而 CQE 中的即时数据（Immediate Data）用于识别不同的请求类型，包括 *RDMA_IMM_DATA_ACK* 和 *RDMA_IMM_DATA_MESSAGE* 类型。生成响应消息后，消息处理程序将利用单边原语（即 RDMA Write 原语）发送。

4.4　系统设计

在本节中，我们将介绍 RDMA 驱动的共识同步管理器、新的节点引导协议和区块广播机制的详细实现。

4.4.1　BoR 消息定义

表 4-3　BoR 系统中面向 RDMA 内存语义的消息类型定义

消息类型	枚举值
时间戳消息（time message）	RM_TIME
通知消息（notice message）	RM_NOTICE_RESPONSE
暂停消息（go away message）	RM_GO_AWAY_RESPONSE
握手消息（handshake message）	RM_HANDSHAKE_REQUEST
请求消息（request message）	RM_CATCHUP_REQUEST
区块同步消息（sync request message）	RM_SYNC_REQUEST

BoR 上的传输数据可以分为四种：*ACK* 确认、*Control Message* 控制消息、*block* 区块和 *transaction* 事务。控制消息由握手消息（*handshake message*）、离开消息（*go away message*）、时间消息（*time message*）、通知消息（*notice message*）、请求消息（*request message*）和同步请求消息（*sync request*）组成。握手消息、离开消息和通知消息用于进行握手请求/响应，以同步接收方和发送方节点之间的区块状态。Time Message 用于节点间的心跳检测。请求消息和同步请求消息分别用于同步可逆区块/事务和不可逆区块/事务。在 RDMA 驱动的消息处理机制中，为了避免频繁注册和注销 RDMA 内存区域（MR）的开销，引入了内存池来管理 RDMA MR。为了提高分配固定大小的 MR 的效率，用于从 MR 池中发送/接收控制消息和相应的远程地址/密钥，将多个不同类型的消息合并到相同的消息定义中，命名为 *RdmaMessage*。我们使用 *RdmaMssage* 数据结构中的字段 *RdmaMessageType* 来区分不同类型的消息。*RdmaMessageType* 的定义如表格 4-3 所示。

4.4.2　RDMA 驱动的同步管理器

RDMA 驱动的同步管理器是一个非常重要的组件，它负责维护 RDMA 通道列表、RDMA 消息缓冲区、基于 RDMA 完成事件通道的异步消息处理、区块/事务同步等。区块/事务同步过程由基于 RDMA 的 P2P 网络中的对等节点之间的握手协商触发。我们假设节点 A 是接收端，B 是发送端。然后节点 A 和 B 之间的握手协商过程如图 4-7 所示。如果 B 接收到的握手请求无效，B 利用内置即时数据的单边 RDMA Write 向节点 A 发回相应原因的 *go away*

消息。否则，B 会返回相关通知消息，并比较 A 和 B 之间的区块头 ID、区块头号、最后一个不可逆区块号。

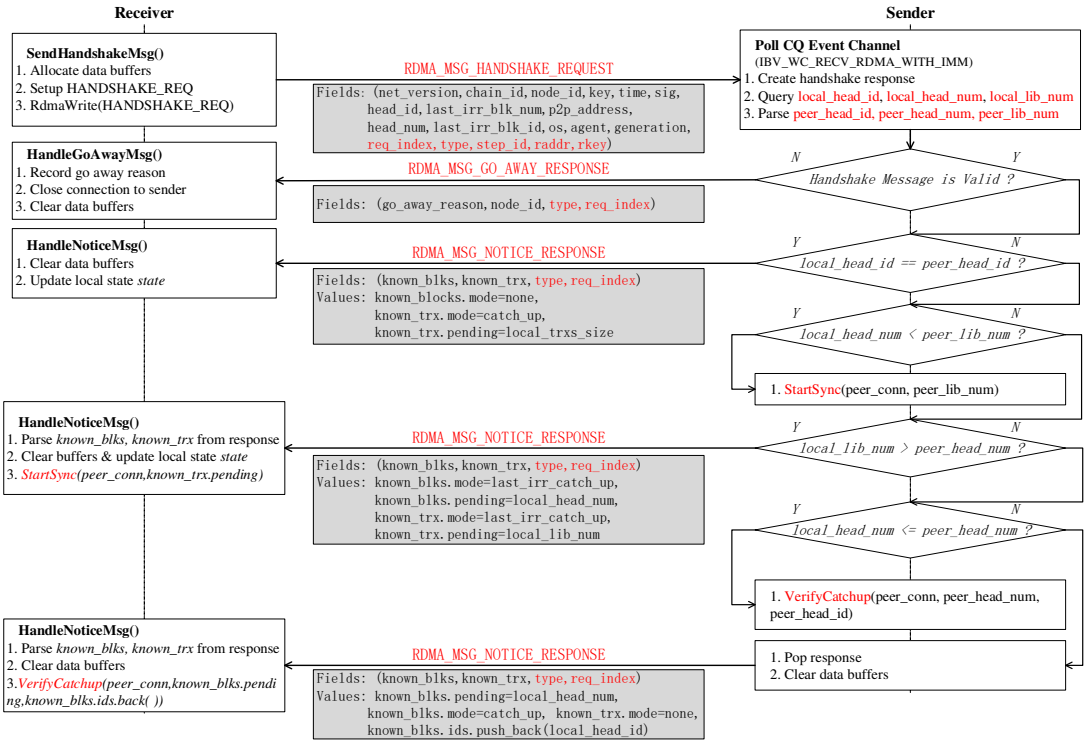


图 4-7　BoR 内基于 RDMA 的区块链通信协议

算法 5 描述了如何使用 BoR 服务器内的 RDMA 事件通道进行异步完成队列（CQ）处理。完成事件通道 *eventChannel* 用于在 CQ 中传递有关即将完成的工作的通知，这可以显著降低 CPU 利用率。保护域 pd 用于保证不同 RDMA 通道之间的资源隔离（如地址句柄和内存区域），而输入 wc 是一个预分配的工作完成数组，它可以在一次调用 *ibv_poll_cq()* 中缓存多个工作完成。

一旦在两个对等 BoR 节点之间建立了面向连接的 RDMA 通道，每个节点中的 BoR 服务器将启动一个 *PollCQ* 线程，以异步侦听附加到来自完成队列的即将到来的工作请求的事件通知。具体来说，在 *PollCQ* 线程的循环体内，附加到 *eventChannel* 的 *ibv_get_cq_event()* 方法被调用并被阻塞，直到传入工作完成的事件（例如完成的发送和接收请求）被触发。然后通过调用 *ibv_ack_cq_event()* 方法确认事件，同时通过 *ibv_req_notify_cq()* 方法请求下一次完成的通知。通过调用 *ibv_poll_cq()* 方法，将一批 RDMA 操作的工作完成缓存到预分配的 *wc* 数组中，可以显著提高 BoR 的并发水平。对于 *wc* 数组中的每个工作完成 *wc[i]*，不同的操作码 *opcode* 代表相应的 RDMA 原语。例如，操作码 *IBV_WC_RECV_RDMA_WITH_IMM* 表示带有即时数据的远程 RDMA Write 操作已完成，有效载荷数据已被推送到本地接收队列，而操作码 *IBV_WC_RDMA_WRITE* 表示本地 RDMA Write 操作已完成。当 *wc[i].opcode* 等于 *IBV_WC_RECV_RDMA_WITH_IMM*，提取即时数据 *imm* 和相关的 RDMA 通道上下文。如果 *imm* 是 *RDMA_IMM_DATA_ACK*，则接收 *ACK* 消息并从发送队列发送下一条 RDMA 消息。如果 *imm* 小于等于 *RDMA_IMM_MAX_REQUEST_ID*，则接收一个区块或事务数据。否则，我们从指定的接收 *MR* 解析控制消息 *rdmaMsg*。根据 *rdmaMsg* 中的 *type* 字段，我们可以识

别不同类型的控制消息，如握手和通知消息，并采取相应的消息处理程序。当 *wc[i].opcode* 等于 *IBV_WC_RDMA_WRITE* 时，提取 *writeType* 标志，用于检查控制或数据消息的 RDMA Write 操作是否完成。

Algorithm 5 基于 RDMA 完成队列事件通道的异步消息处理

Require: *eventChannel*, *pd*, *wc*

eventChannel: Completion event channel, to wait for work completions.

pd: RDMA protection domain

wc: Pre-allocated work completions array used for polling

Ensure: *None*

```
1:  isRun ⇐ true
2:  cq ⇐ Completion queue, to poll on work completions
3:  cqContext ⇐ CQ Context
4:  while isRun is true do
5:      ibv_get_cq_event(eventChannel, cq, cqContext)
6:      ibv_ack_cq_event(eventChannel, cq, cqContext)
7:      ibv_req_notify_cq(cq, 0)
8:      ne ⇐ ibv_poll_cq(cq, sizeof(wc), wc)
9:      for i in range(0, ne) do
10:         if wc[i].opcode is IBV_WC_RECV_RDMA_WITH _IMM then
11:             rdmaChannel ⇐ wc[i].wr_id
12:             rdmaChannel.Recv()
13:             imm ⇐ wc[i].imm_data
14:             if imm is RDMA_IMM_DATA_ACK then
15:                 receiveAckMsg()
16:                 sendNextItem()
17:                 continue
18:             else if imm<=RDMA_IMM_MAX_REQUEST_ID then
19:                 recvBlockTransactionContent()
20:                 continue
21:             end if
22:             rdmaMsg ⇐ parseMsgFromMR()
23:             type ⇐ rdmaMsg.msgType
24:             if type is RDMA_MSG_HANDSHAKE_REQ then
25:                 handleHandshakeMsg()
26:             else if type is RDMA_MSG_GO_AWAY_RES then
27:                 handleGoAwayMsg()
28:             else if type is RDMA_MSG_NOTICE_RES then
29:                 handleNoticeMsg()
30:             else if type is RDMA_MSG_SYNC_REQ then
```

```
31:             queryLocalIrreversibleBlock()
32:             rdmaWriteIrreversibleBlock()
33:         else if type is RDMA_MSG_VERIFY_CATCHUP then
34:             queryReversibleBlockOrTxn()
35:             rdmaWriteReversibleBlockOrTxn()
36:         end if
37:     else if wc[i].opcode is IBV_WC_RDMA_WRITE then
38:         wrId ⇐ wc[i].wr_id
39:         writeType ⇐ wrId.write_type
40:         if writeType is RDMA_WRITE_ID_MSG then
41:             sendNextItem()
42:         else if writeType is RDMA_WRITE_ID_ BLOCK_WRITE then
43:             popResponse()
44:         end if
45:     end if
46:   end for
47: end while
```

4.4.3 基于 RDMA 的新节点引导协议

对于区块/事务同步状态，可以分为三种类型：*catch_up*、*last_irr_catch_up* 和 *normal*。对于区块，*catch_up* 状态意味着发送方 B 将所有本地可逆区块打包发送给接收方 A；*last_irr_catch_up* 状态表示发送方 B 打包所有本地不可逆区块并发送给接收方 A；正常状态意味着根据接收方 A 中的区块 id 向量在发送方 B 中搜索不可逆区块，并将查询到的区块发送给节点 A。而对于事务，*catch_up* 状态意味着发送方 B 将节点 A 所需的可逆事务 ID 作为输入，将查询到的事务发送给节点 A；*normal* 状态意味着发送方 B 将节点 A 所需的不可逆事务 ID 作为输入，并将查询到的事务发送给节点 A。

当新节点 N 刚加入区块链网络时，节点 N 中的本地区块头号小于其对等节点中最后一个不可逆区块头号（节点 N 处于 *last_irr_catch_up* 状态）。这会驱动节点 N 开始同步来自对等节点的不可逆区块，如图 4-9 所示。一旦节点 *N* 中的本地区块头编号大于对等节点的最新不可逆头编号且小于对等节点（节点 *N* 处于 *catch_up* 状态），节点 *N* 将开始同步来自目标对等节点的可逆区块，直到本地头号大于或等于对等节点中的头号（节点 *N* 处于正常状态）。区块/事务同步过程由具有即时数据的单边 RDMA 写原语驱动，如图 4-8 和图 4-9 所示。

4.4.4 RDMA 混合原语和资源分配

本节介绍了 BoR 的其他优化注意事项，包括使用混合 RDMA 原语和高效的 RDMA 资源分配。

混合 RDMA 原语。正如之前的工作[89] 所指出的，混合 RDMA 原语的使用可以为基于

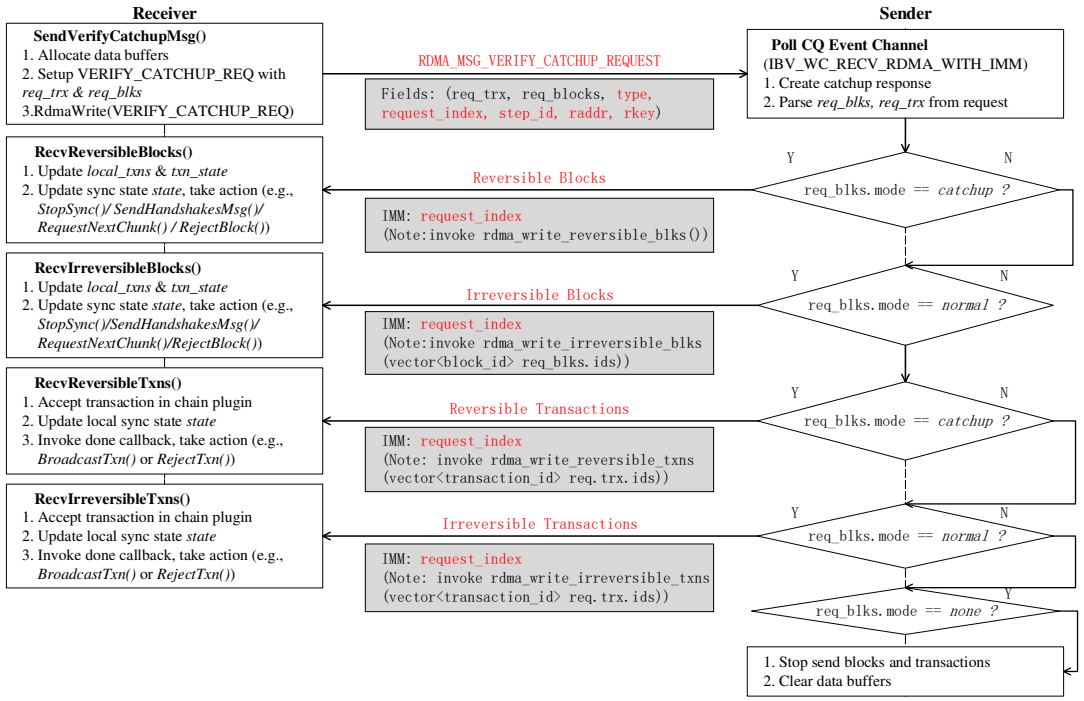


图 4-8　BoR 中基于 RDMA 的 *verify_catchup()* 过程

RDMA 的分布式事务带来更好的性能增益。从中学习，我们利用双边 RDMA Send/Recv 原语在互连节点之间交换预先注册的 TX/RX MR，并使用内置即时数据的单边 RDMA Write 原语来加速时延敏感的控制消息传输和带宽敏感的数据消息传输。

RDMA 资源管理。 RDMA 资源的不合理管理可能会导致 RNIC HCA（Host Channel Adapter）内的缓存未命中，这会显著影响 BoR 的整体性能。为了最大限度地发挥 RDMA 的性能优势，我们对消息缓冲区、发送/接收队列（Queue Pair，QP）和完成队列（Completion Queue，CQ）采取了不同的调优策略。例如，BoR 的消息缓冲区与缓存行大小对齐，以消除读-修改-写。为了最大限度地减少 RNIC 和 CPU 中的转换后备缓冲区（Translation Lookaside Buffer，TLB）未命中，BoR 使用 2MB 页面大小的大页面在关闭的 NUMA（Non-Uniform Memory Access）节点上分配消息缓冲区。利用共享接收队列（Shared Receive Queue，SRQ）来提高 BoR 的可扩展性，同时利用每个线程/核心的单个完成队列来避免对多个内存段进行轮询。为了实现更低的 CPU 消耗，BoR 使用完成事件通道并在一次调用中缓存多个工作完成。此外，利用预注册的 MR 池进行 BoR 的单边转移，以减轻注册或取消注册 MR 引入的时延开销。

4.4.5　应用案例：BoR 驱动的发布订阅模型 BPS

今年来，发布/订阅流系统越来越多地被用作多租户云中的上游中间件层，预处理并输送海量数据到下游的数据分析和数据仓库等系统。面向多租户的云服务中，多个不受信任的应用或租户会公用同一个发布/订阅系统，这带来了验证的安全挑战，包括隐私敏感的数据/元数据访问威胁和关键元数据被未授权篡改等。受基于 RDMA 的高性能区块链特性启发，我们进一步面向多租户云服务设计与实现了一种 BoR 增强的、面向多租户的、去中心

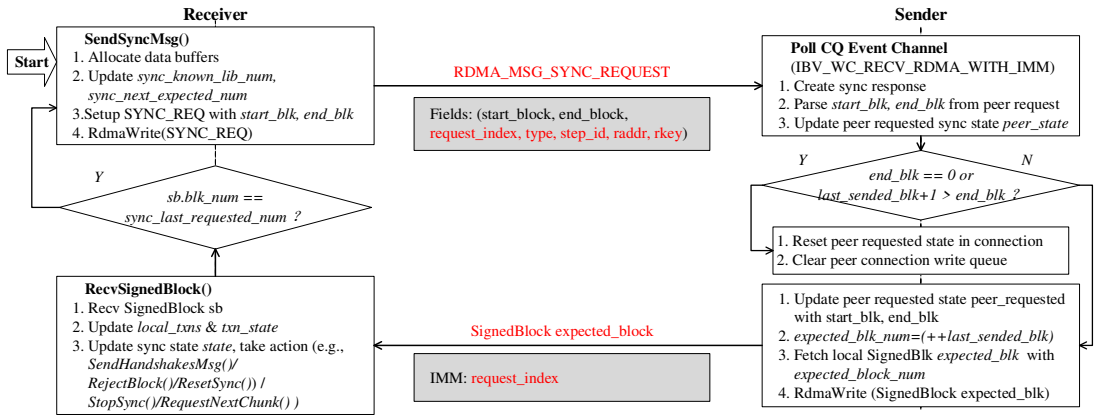


图 4-9　BoR 中基于 RDMA 的 *start_sync()* 过程

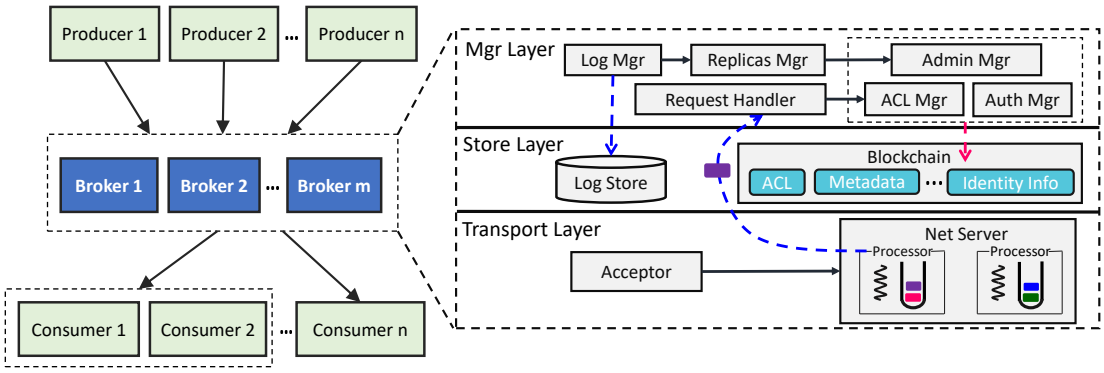


图 4-10　基于 BoR 的发布订阅模型 BPS 架构概览

化的主题发布/订阅模型 BPS。

为了在共享租户模型中实现安全高效的发布/订阅流服务，我们在设计实现 BPS 时需要在安全性、可靠性和高性能之间进行权衡。也就是说，我们必须确保跨多个租户的可靠机密性、匿名性、身份验证和授权，同时最小化性能开销，从而为基于主题的发布订阅系统提提升端到端消息吞吐、并降低主题消息的访存时延。具体设计目标如下：

- **数据完整性**。要求利用必要的数据校验和审计来确保主题消息和关键元数据的完整性。
- **查询效率高**。可以跟踪和审计特定主题的所有操作（如注册、查询、发布或订阅），而无需遍历整个区块链。
- **入侵容错**。即使某个代理（Broker）由于恶意攻击（例如服务拒绝攻击）而崩溃，该代理服务的发布者和订阅者也可以被其他代理接管，并且代理上存储的元数据或消息分区在其他代理上有可靠的副本。

作为一个 BoR 增强的基于主题的发布/订阅流系统，BPS 包含三个主要组件：代理（Broker）、生产者（Producer）和消费者（Consumer）。*Producer* 通常代表不同的租户，负责采集或生成不同的数据集，并同步到 *Broker* 端进行处理。*Consumer* 通常对应租户使用的下游流处理系统（如 Spark、Tensorflow），从而进行数据挖掘和推理计算。作为 BPS 的核心组件，

表 4-4　BoR 驱动的 BPS 系统与已有发布/订阅系统的比较

特性	无 Broker 且基于内容的 Pub/Sub 系统				Broker 驱动且基于话题的 Pub/Sub 系统				
	Tariq [126]	Anusree [127]	Shitole [128]	AnonPubSub [129]	Ion [130]	AKPS [131]	SPS [132]	Kafka [133]	BPS
机密性	✓	✓	✓	✓	✓	✓	✓	✓	✓
匿名性	✗	✗	✗	✓	✓	✓	✓	✓	✓
授权管理	✓	✓	✓	✓	✓	✓	✓	✓	✓
性能	✓	✓	✓	✗	✗	✓	✗	✓	✓
完整性验证	✓	✓	✓	✓	✗	✗	✓	✗	✓
多租户隔离	✗	✗	✗	✗	✗	✓	✗	✓	✓
防篡改	✗	✗	✗	✗	✗	✗	✓	✗	✓
去中心化安全性	✗	✗	✗	✗	✗	✗	✓	✗	✓

Broker 以主题（*Topic*）的形式管理来自上游的消息数据，同时被动地将主题分类的消息转发给租户依赖的下游 *Consumer*。换句话说，*Producer* 以推送（*push*）的方式（即 *Produce*）向 *Broker* 发布主题指定的消息，而 *Consumer* 订阅分配的主题并以拉取（*pull*）的方式（即 *Fetch*）消费消息。各个租户之间的消息数据隔离是通过基于主题的分组和基于 BoR 区块链的细粒度访问控制来实现。BPS 架构概览如图 4-10 所示。相比于完美的发布订阅系统 Kafka，BPS 可以以最小的性能开销实现更安全高效的通信模型，表 4-4 总结对比了 BPS 与其他发布/订阅系统的差异和优势。

4.5　系统性能评估

4.5.1　实验环境与方法

表 4-5　实验评估环境

服务器数量	4
服务器型号	PowerEdge R730
CPU 核数量	48
内存	128GB DDR4
操作系统版本	CentOS 7.4.1708
RDMA 以太网交换机	Mellanox SN2100 (100Gbps)
RDMA 网卡	MCX516A-CDAT ConnectX-5 (100Gbps)

硬件环境配置。我们在基于 RoCEv2 网络上进行了 BoR 和传统 TCP EoS 之间的性能评估分析，测试台上有 4 台服务器，通过 Mellanox SN2100 交换机互联，如表格 4-5 所示。每台服务器的 DRAM 大小为 128 GB，DDR4 类型，速度为 2,400 MHz。每台服务器配备两个

E5-2687Wv4 CPU 处理器，具有 12 个 CPU 内核和 3.00 GHz 频率。每个 CPU 内核的 L1 缓存为 768 KB，而 L2 缓存为 3,072KB。一个 CPU 处理器上的 12 个内核共享相同的 30 MB L3 缓存。我们服务器中的 RNIC 是 ConnectX-5 MCX516A-CDAT 100 Gbps RoCE NIC，并通过 PCIe 3.0x16 连接到 SN2100 交换机。Nodeos 和 keosdare 部署在服务器 192.168.2.1 中，而三个生产者部署在服务器 192.168.2.2、192.168.2.3 和 192.168.2.4 中。所有节点的操作系统版本为 CentOS 7.4.1708，OFED 版本为 v4.5-1.0.1。

区块初始同步的实验设计。我们分别在 A、B、C 三个主机上部署 BoR 和原始 EoS 区块链网络测试平台，然后从 EosNode 测试中获得 5 个不同规模（10、20、30、40 和 50 GB）的历史区块数据集网络[134]。我们基于 5 个区块文件设置了 5 个案例。在每种情况下，相应的历史区块文件都会在 BoR 和传统 TCP EoS 系统上回放。我们假设机器 D 作为分布式共识网络的新节点。

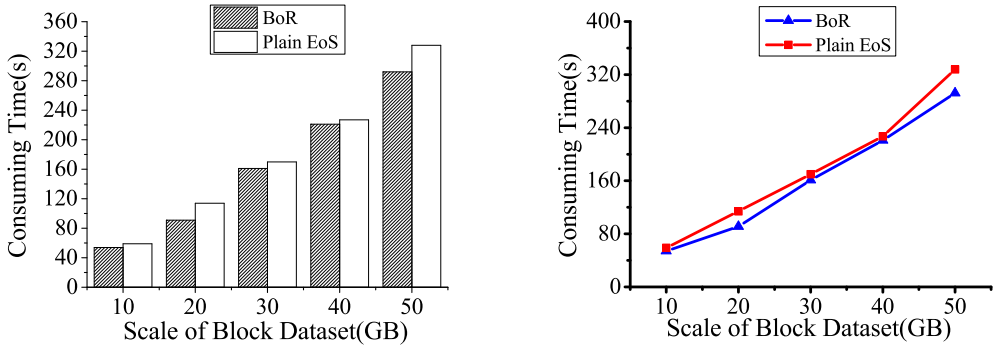


图 4-11　不同规模的历史区块数据集下区块链系统初始化同步的耗时比较

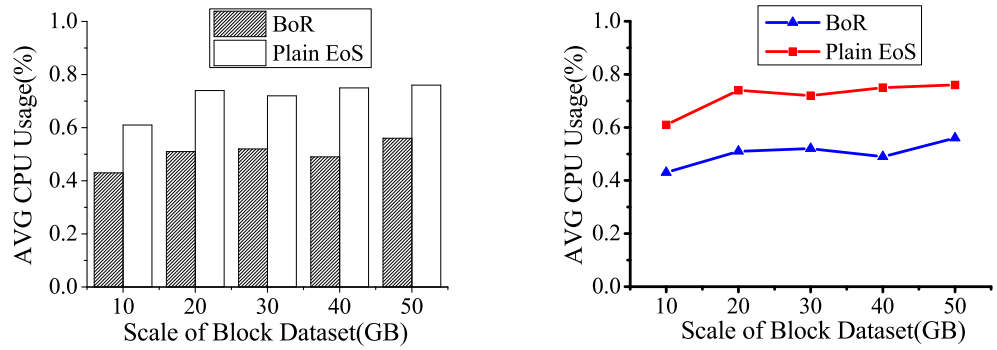


图 4-12　BoR 和原始 EoS 区块链系统在递增的区块数据集规模下平均 CPU 占用率的分布

4.5.2　初始区块同步的性能分析

对于 BoR 和传统 TCP EoS 之间的性能比较，我们集中在两个性能指标上，包括共识消耗时间和 *CPU* 占用率。消耗时间定义为完成区块链网络中新节点的初始区块同步所需的时

间。历史区块数据集的规模被视为一个自变量。CPU 占用率表示在初始区块同步期间产生的 CPU 开销。它包含三个维度：CPU 利用率的时间序列分布、不同规模区块数据集的平均 CPU 占用率以及 BoR 和传统 TCP EoS 之间 CPU 占用率差异的峰值。Linux 性能监控工具 *sar* 用于以时间序列方式每秒收集一次 CPU 利用率。平均 CPU 占用率（即 AVG CPU Usage）的计算方法是将累积的 CPU 占用率值除以总消耗时间。所谓 *CPU D-value* 的差异，是指 BoR 和传统 TCP EoS 在分布式共识过程中的 CPU 占用率的差距。峰值代表 *CPU D-value* 的最大值。分布式共识协议的每条测评结果均取多次测量的平均值。

根据性能结果，我们观察到 RDMA 增强范式可以将初始区块同步的时延降低高达 20.2%，并将平均 CPU 开销降低 26.4%~33.9%，这可以促进 BaaS 云数据中基于 DPoS 的高性能区块链。

1. 初始区块同步的处理时间

图 4-11 描述了在 BoR 和传统 EoS 网络中新节点初始同步期间，不同规模历史区块数据集下的处理消耗时间对比。我们可以发现，与基于 TCP 的传统 EoS 相比，采用单边 RDMA Write 的 BoR 实现了更低的整体消耗时间（高达 20.2%），最大时延间隙为 36 秒，最小时延间隙为 5 秒。在最好的情况下，BoR 的初始化区块是同步的，BoR 共识确认的整体耗时相比于原生 EoS 降低可达 20.2%（如图 4-11(a) 所示）。我们可以在图 4-11(b) 中发现，一旦区块记录的规模大于 40 GB，BoR 与传统 TCP EoS 之间的共识耗时差距会显著增加。这得益于面向商用 RDMA 网络的高性能分布式共识协议，减少了区块数据在共识网络广播传递中的数据路径内存拷贝的次数，从而显著降低了多节点之间的区块同步时延。同时 BoR 采用基于 RDMA 完成事件的异步通信模型，交叉重叠区块数据传输和共识计算过程，进一步降低了区块数据通信引入的 CPU 开销，提升了分布式共识效率。

2. 初始区块同步的 CPU 开销

CPU 利用率和平均 CPU 利用率的时间序列分布如图 4-13 和 4-12 所示。与传统 TCP EoS 相比，BoR 显著降低了 CPU 开销。从 CPU 占用率分布来看，从图4-13可以看出，在不同的区块大小下，BoR CPU 占用率的趋势线一般都低于传统 TCP EoS 的趋势线。从图 4-12(a) 中我们可以看到，BoR 的 CPU 利用率相比于传统 TCP EoS 降低了 26.4~33.9%。在最好的情况下，BoR 的平均 CPU 占用率为 0.49%，而 EoS 的平均 CPU 占用率为 0.75%。而且，随着区块记录规模的增加，RDMA 驱动的 BoR 的平均 CPU 占用率比传统 TCP EoS 更稳定，如图 4-12(b) 所示，这可以带来稳定的 BaaS 质量。BoR 带来的性能收益主要得益于 RDMA 驱动的高性能分布式共识协议设计，预注册的 RDMA 内存池和基于 RDMA 完成事件通道的异步通信模型也进一步降低了系统的 CPU 开销。同时，我们利用 RDMA 共享接收队列来缓解高并发区块同步负载下的 RDMA 硬件缓存资源争抢，改进基于 RDMA 的分布式共识协议的扩展性。

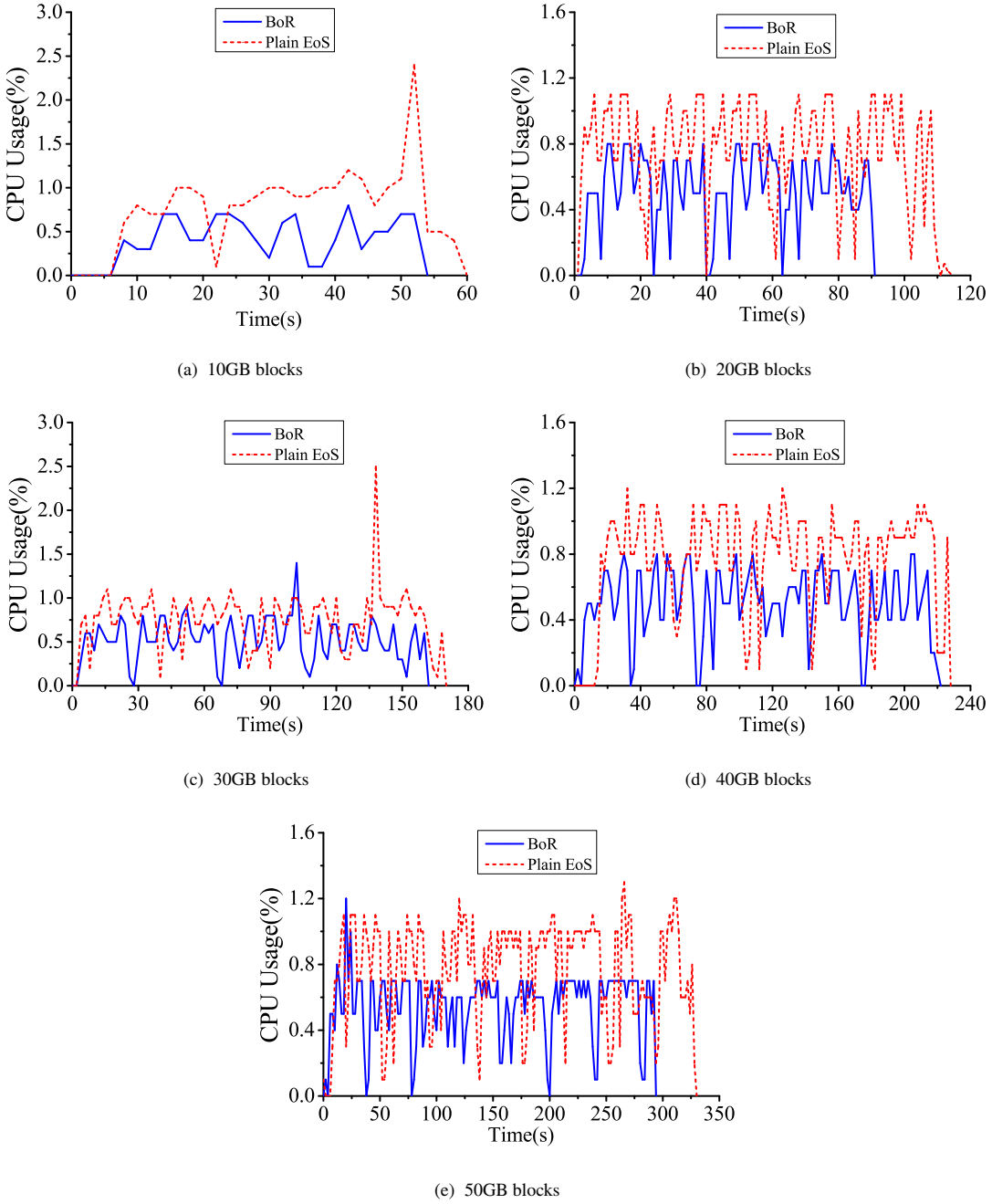


图 4-13　初始区块同步在不同历史区块数据集合的负载下 CPU 利用率

4.6　本章小结

在本文中，我们首先分析了新兴 BaaS 云中时延/CPU 敏感的分布式共识协议的性能瓶颈。根据我们的调研分析，分布式共识是传统区块链（如比特币、以太坊）的系统性能瓶颈。幸运的是，分布式共识协议近年来得到了显著的改进，在事务吞吐方面大幅提升。然而，随着分布式共识协议的快速演进，基于传统 TCP/IP 网络的区块/事务传输的性能瓶颈逐渐显露出来。因此，我们提出了 BoR，一个面向商用 RDMA 网络的高性能分布式共识协议，它利用 RDMA 通信原语实现高吞吐量和低时延，无需 CPU 干预，用于 BaaS 云中高并发下

的初始区块同步和区块/事务传播。支持 RDMA 网络中的内核绕过和零拷贝技术被 BoR 用来有效地实现更高的性能。RDMA 完成事件通道还用于异步接收控制消息或区块/事务，同时使用内置即时数据的单边 RDMA Write 原语来发送控制和数据消息。为了缓解高 CPU 开销和长时间消耗，BoR 在两种范式中利用 RDMA：一种是新节点加入 P2P 网络时的初始同步，另一种是生产者生成新区块时的区块广播。我们的评估表明，BoR 显著优于原始的 EoS 区块链系统。与最先进的 EoS 区块链相比，BoR 明显降低了高达 20.2% 的时延，并在新节点加入区块链网络时降低了 26.4~33.9% 的 CPU 利用率。

我们未来的工作可能会扩展 BoR 以支持多路径 RDMA 传输，以便在 BaaS 云数据中心进行初始区块同步。我们还计划利用可靠数据报传输进行 RDMA 通信，以提高基于 BoR 的 BaaS 云的可扩展性。除此之外，我们的目标是将新兴的高速非易失性内存（Non-Volatile Memory，NVM）与商用 RDMA 网络协同设计，以加速面向商用 RDMA 网络的分布式共识协议的区块数据持久化性能。

第 5 章　NTSocks: 基于 PCIe 互连的高性能用户态协议栈

本章的主题是设计基于 PCIe NTB 新硬件的轻量级协议栈，在本文中所处位置如图 5-1 所示。本章从软硬件协同设计的角度，面向高性能机架规模通信，将提出一种基于 PCIe NTB 新硬件互连的控制平面与数据平面分离的高性能用户态协议栈 NTSocks，通过一个用户级间接层屏蔽底层 PCIe 互连低级抽象的同时，实现兼容、多核扩展、性能隔离的网络功能，弥补了商用 RDMA 在机架内通信中存在的 PCIe 协议与网络协议转译开销的不足，自下而上地增强不同类型系统的性能。

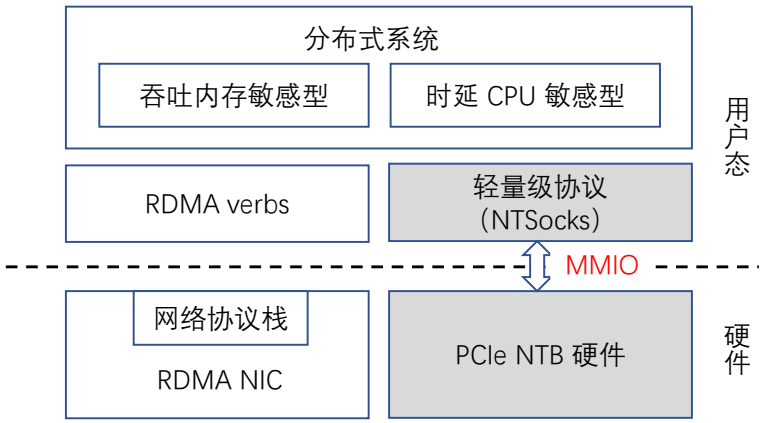


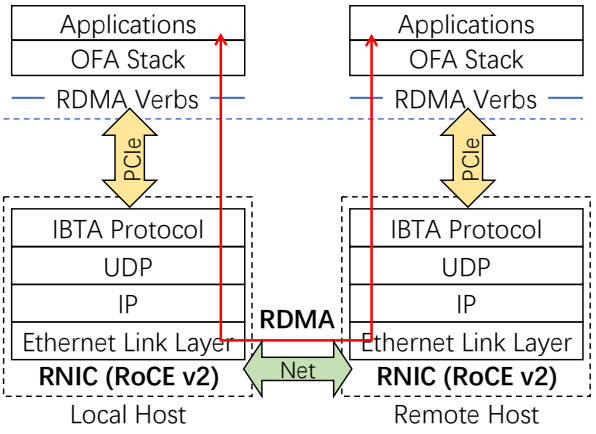
图 5-1　本章主题：基于 PCIe NTB 新硬件的轻量级协议栈设计，用灰色背景的方框标记

5.1　引言

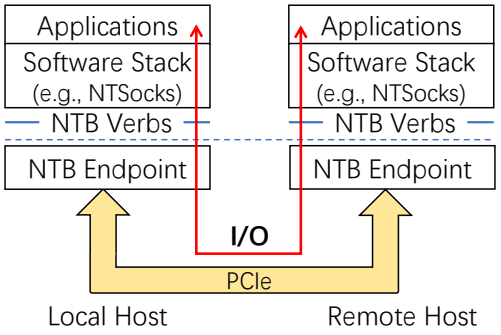
数据中心内超低时延和高吞吐的机架规模网络已经成为提供高质量数据中心服务的基础，并且时延越低越好[4, 6, 26, 60]。一个明显的趋势是，高密度的新型计算（如 TPU[135]）和存储（如非易失内存 NVM[136]）硬件部署到一个机架内，称之为机架级计算机（rack-scale computers）[5]。这促使潜在的系统瓶颈从计算转移到网络[62, 137]。然而，对于机架规模网络的优化和加速，基于以太网的研究工作[59, 65, 138, 139] 分为两大类：一部分工作是依赖于特定工作负载，这限制了网络架构的泛化能力且性能仍有很大提升空间；而另一部分工作是依赖于中心化的控制器，这损坏了网络系统的性能。

因此，近年来的许多研究致力于内核旁路网络，即利用协议栈硬件卸载（如 RDMA）[10, 12] 或者用户空间网络 I/O 技术（如 Intel DPDK）[11] 来大幅降低时延。但是，这些解决方法仍然依赖于厚重且冗余的分层协议栈，从而不可避免地引入了协议层之间的协议转译开销。例如，尽管融合以太网上的 RDMA 实现（即 RoCE）允许网卡直接访问远端机器的内存，一

次单向数据传输仍然需要在 PCIe 总线、RDMA 传输协议（即 IBTA protocol）、UDP 之间至少 4 次的协议转译，如图 5-2(a) 所示。因此，不可忽略的时延开销被引入。同时，对机架内跨机器之间 PCIe 连接的外围设备（如片上系统）基于以太网的访问，进一步增加了数据路径上协议转译的次数。



(a) RDMA 网络栈



(b) PCIe NTB 栈

图 5-2　相比于商用 RDMA 网络栈，PCIe NTB 避免了 PCIe 与网络协议之间的转译开销

那么是否可以摆脱机架内网络的协议转译开销来进一步降低时延呢? 我们认为使用 PCI Express（PCIe）互连作为机架内高速网络技术是一个理想的选择，因为这不需要 PCIe 总线和传输协议之间的转译，如图 5-2(b) 所示。具体而言，尽管 PCIe 的设计初衷是连接多个 PCIe 设备到同一个 CPU 为中心的计算机系统[140]，PCIe 非透明桥（Non-Transparent Bridge，NTB）作为一个 PCIe 域间通信的特殊设备，使得多个独立的机器互连到同一个 PCIe 网络成为可能[23, 141]。由于总所周知的系统内核空间实现（如 Intel NTB 内核驱动 *ntb_hw_intel*）[142] 的低效（如高昂的系统调用开销）[11, 143]，我们聚焦在使用用户态的 NTB 驱动如 DPDK 轮询模式驱动（*Poll-Mode Driver*, PMD）[13] 来消除内核的复杂性。通过使用事务层协议（Transaction Layer Protocol, TLP）来在独立的计算机系统之间映射共享的内存空间，PCIe NTB 允许应用以极低的时延（相比 RDMA 实现 2.3~5.6 倍的加速，如图 5-3 所示）和高带宽（接近 PCIe 带宽理论值）来访问远程机器的内存。

然而，由于 PCIe NTB 本来是为跨 PCIe 域空间的设备间通信（如设备共享[23]）而设计的，直接将它用于机架内的网络通信会遇到 3 个主要挑战。

第一，在原生 NTB（即用户态 NTB）与上层应用之间**存在抽象不匹配问题**。不同于基

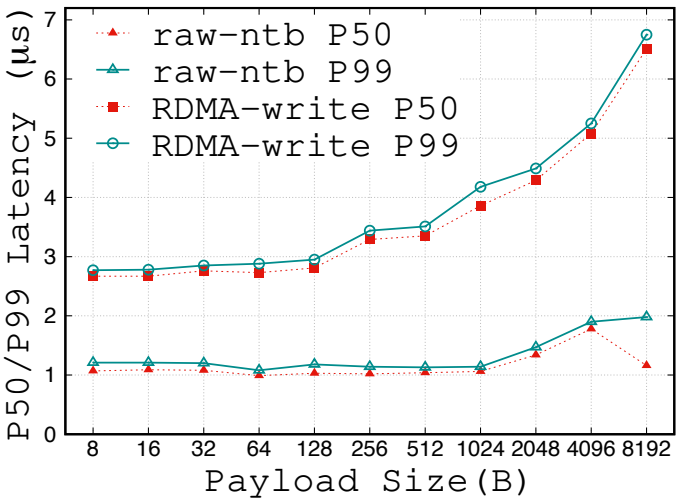


图 5-3　原生用户态 NTB 传输可以实现比 RDMA （ConnectX-5 NIC）更低的时延

于以太网的 TCP/IP，原生 NTB 缺少通用的网络功能抽象（如寻址、路由），仅提供接近硬件原语的低级抽象。对于预期的抽象，提供编程接口兼容性的同时保留原生 NTB 的性能收益是非常有挑战的。

第二，原生 NTB **缺少可扩展的数据平面**来应对并发流量负载。这是因为原始的设备驱动有一个默认假设，即驱动对 NTB 设备拥有独占式的控制[23]。而且，由于 NTB 硬件中地址转译的复杂性，映射出的共享内存空间是有限制的，例如 Intel Xeon 处理器上仅支持 512MB[13]。这迫切需要一个新的数据平面设计，在实现高效地复用 NTB 传输链路的同时支持多核扩展性。

第三，现在的 NTB 架构**缺少应用之间的性能隔离**。由于不同的应用有不同的流量模式，NTB 资源复用不可避免地引入了性能干扰如线头阻塞(Head-Of-Line blocking, HOL blocking)和复用单元之间的负载倾斜，特别是在处理动态的工作负载时（如随机断连、变化的流速率)。

为此，本章设计了 **NTSocks**，首个构建在 PCIe NTB 互连上的用户空间机架级网络架构，可以通过软硬件协同设计为机架规模的系统提供高性能的网络功能，同时满足了兼容易用性、多核扩展性和多租户性能隔离的设计目标。NTSocks 的核心包括：一个间接代理（即数据平面），以独立进程的形式运行在每个机器上，并且轻量级地虚拟化 NTB 传输资源以实现多应用之间的资源共享；一个分离的软件监控器（即控制平面），实现类似以太网的控制路径功能。这种数据平面与控制平面分离的架构不仅可以从数据路径（如数据流）上移除控制路径（如寻址和路由），从而预留原生 NTB 的性能收益；而且可以很容易地支持故障转移（如 TCP 回退），在无需户级应用程序停机的情况下实现透明升级 NTSocks 系统。

基于以上架构，我们进一步提出了三个关键设计以更好地权衡不同的设计要求，解决上述挑战。具体而言，NTSocks：

- 通过 3 种方法设计了**一个支持通用网络功能的、高性能的类似套接字的抽象**：（1）使用 NTB *Remote Write* 原语实现一个基于 NTB 共享内存的无锁环形缓冲区（第 5.4.1 节）；（2）透明的零拷贝支持（第 5.4.5 节）；（3）一个自适应的接收端驱动的流量控制方法来阻止消息溢出（第 5.4.4 节）。

- 提出了**一个数据平面核心-分区模型**（即 *core-partition* 模型）（第 5.4.2 节），可以为每个分区按需分配 CPU 核心，从而在多核扩展性和 CPU 使用效率之间达到更好的权衡。分区是一个实现多核机器上扩展性的新抽象，它将有限的 NTB 共享内存空间划分为多个并行单元，每个单元（即一个分区）是核心驱动的，且被一组连接复用。基于 NTB 门铃（doorbell）寄存器的接收（Receive，*RX*）事件通知进一步被用来避免 *RX* 轮询。

- 在分区抽象的基础上设计了**一个层次化性能隔离机制**（第 5.4.3 节），即：（1）分区内面向每个连接的消息切片，用于消除一个共享的分区内多个连接之间的线头阻塞；（2）分区间连接粒度的负载均衡，包括基于静态工作负载（如每个分区中活跃连接的数量）的 *Round-Robin* 连接分发和基于动态工作负载的连接热迁移。

我们在 DPDK 轮询模式驱动之上实现了 NTSocks。我们发现 NTSocks 以可接受的开销实现了优于已有完美网络栈的性能，同时实现了高扩展性和性能隔离（第 5.6 节）。例如，相比 Linux Socket 和 *libvma*[144], NTSocks 降低时延分别到 1/32 和 1/4.4，降低尾时延分别到 1/39 和 1/5。我们进一步在没有代码改动或最小化代码改动的情况下，将典型的键值存储系统（Key-Value Store, KVS）、*Nginx* HTTP 服务器和 *Apache* 测量工具（Apache benchmarking tool, ab）移植到 NTSocks 系统上。通过用典型的 YCSB（Yahoo! Cloud Serving Benchmark）工作负载来度量键值存储系统，结果显示，相比于 Linux TCP 和 RDMA 套接字，NTSocks 可以降低键值存储系统的端到端时延分别到 1/24.5 和 1/1.58。对于 Nginx，NTSocks 性能优于 Linux TCP 高达 6.7 倍。我们将会在 https://github.com/NTSocks 开源 NTSocks。

NTSocks 不仅可以通过混合的 PCIe NTB 和 RDMA 部署为主流的数据中心内以太网带来时延的降低，同时对于裸机云[137] 中未来的机架内基于片上系统（system-on-chips，SoCs）的网络形式也保留了扩展友好性（第 5.7 节）。

5.2　背景与动机

本节首先介绍了机架规模系统和网络的问题，接着介绍了 PCIe NTB 的原理和特性。最后阐述了 PCIe NTB 在机架规模系统中的潜力和局限性，也是促使我们设计 NTSocks 系统的动机。图 5-4 显示了原生用户态 PCIe NTB 协议栈的架构以及对上层应用程序提供的编程接口抽象。

5.2.1　机架规模的系统与网络

1.　机架规模系统

数据中心内机架规模系统希望提供低时延和高吞吐量，以满足在线服务或实时分析的严格的 SLO （Service-Level Object）[5, 6, 60]。代表性系统包括键值存储系统[145, 146]、分布式事务系统[88, 147]、分布式共享内存系统[44, 148, 149]、在线深度神经网络推理系统[62, 150, 151]、实时流处理系统[152, 153]，图存储系统[154, 155]，无服务器计算系统[156, 157] 等。

高密度现代硬件的快速发展需要高性能的机架级通信。为了应对不断增长的横向扩展需求，我们认为将高密度现代硬件放入机架是必然趋势[5]。最近的机架级计算机（例如 Intel RSA[158]、TPU Pods[135]、FireBox[159]）或存储（例如 Facebook Lightning[160]、EMC 的 DSSD[161]、

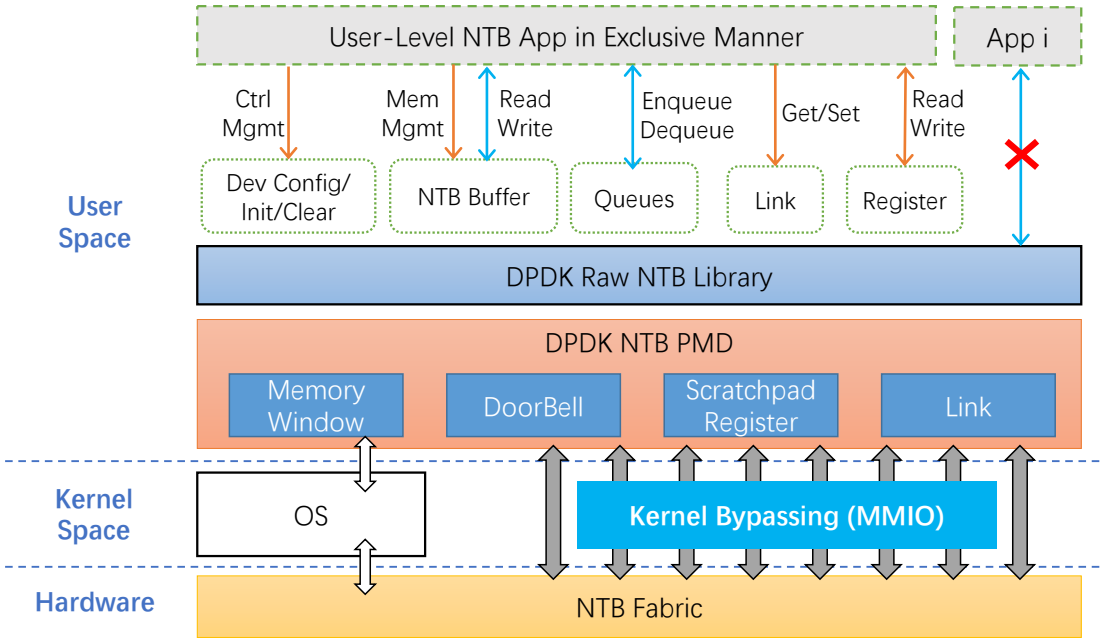


图 5-4　原生用户态 NTB 栈

Decibel[63]）证明了这一点。最近的工作[5, 162]进一步利用可编程交换机来增强机架级系统。上述研究工作的创新对更快、更高效的机架级通信提出了新的挑战[59, 62, 64, 65]。

2. 机架规模网络

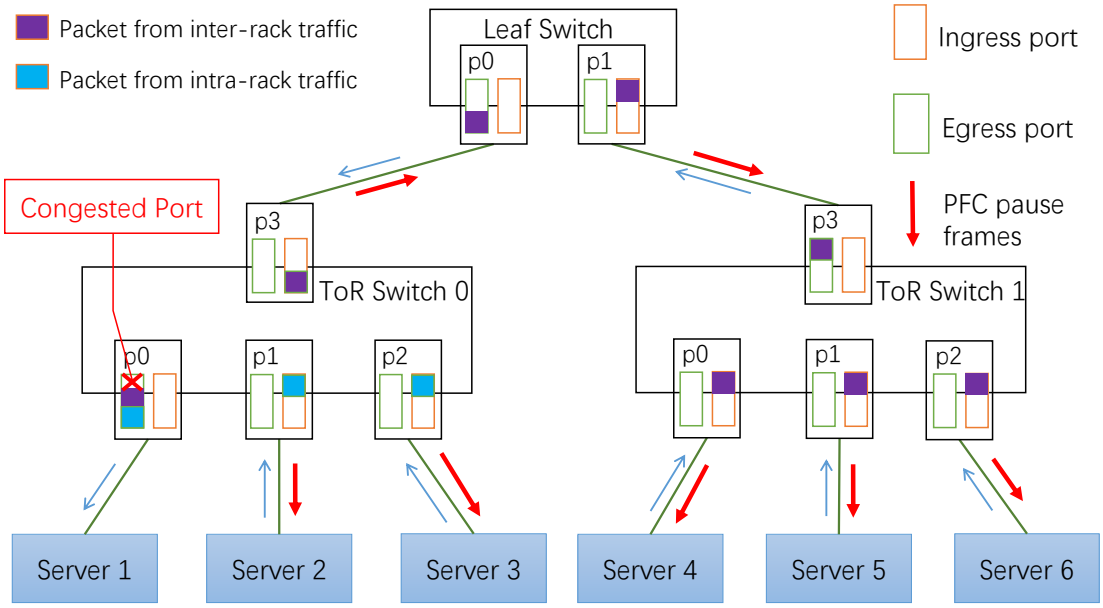


图 5-5　RDMA ToR 交换机出端口 Incast 拥塞问题

为此，近年来商用 RDMA 网络硬件已被广泛应用于数据中心以增强机架级通信[4, 60, 62]。例如，Pangu[4] 引入生产级 *Podset* 内的无损 RDMA 来加速面向多角色的云存储系统。

机架级系统通常会处理大量混合的机架内/机架间的并发请求。典型的模式包括 *fanout* 扇出响应[163]、*all-reduce* 操作[151] 和分布式 *join*[152]。例如，ByteDance 中的 BytePS[151] 在分

布式深度神经网络训练期间聚合了来自机架内/机架间工作节点的模型梯度。

然而，当前的 RDMA 技术在上述的混合 RDMA 流量下可能会导致交换机出口 Incast 拥塞问题和性能下降。当前 PCIe GEN 3×16 （BW_{pcie}）的带宽（Bandwidth，BW）容量为 *128 Gbps*，而主流的 ToR RDMA 交换机（BW_{tor}）的带宽为 *100 Gbps*。在实际数据中心部署中，更多采用的是 25/40/50Gbps 的 ToR 交换机，无法匹配 PCIe BW。如图 5-5 所示，机架内服务器 *2*、*3* 和机架间服务器 *4*、*5*、*6* 同时向目标服务器 *1* 发送数据。所有到目标服务器 *1* 的 RDMA 流量可以抽象为两种流：机架之间的流量（Inter-Rack Flow）、机架内的流量（Intra-Rack Flow）。我们假设：（1）服务器 *1* 中的 PCIe *BW* 在没有本地主机流量（如 RNIC 环回流量）的情况下相对空闲。（2）机架内/机架间流量的总体发送速率 BW_{intra} 和 BW_{inter} 分别为 *50 Gbps* 和 *55 Gbps*。第 5.6.4 节中的实验表明，混合机架间/机架内流量（*105 Gbps*）情况会导致 ToR 交换机 *0* 的出口端口 *p0* 上的 *2:1 Incast*。由于交换机默认的是 DCQCN[12] 算法，检测到了大量的拥塞通知数据包（Congestion Notification Packet，CNP），机架内/机架间的流完成时间增加了约 3 倍（如图 5-27 所示）。我们得出结论，只要满足约束 $BW_{tor} < BW_{intra} + BW_{inter} < BW_{pcie}$，且 BW_{intra} 和 BW_{inter} 均小于 BW_{tor}，上述 Incast 拥塞问题就很容易触发。

尽管先前的工作[10, 12, 164] 使用面向发送率的被动限速方法来缓解上述的 Incast 拥塞问题，但在上述约束条件下仍很难饱和目标服务器中的 PCIe 带宽，并且在严格的服务级目标约束下，上述方法引入的性能损失几乎是不可接受的。

5.2.2 PCIe NTB 及其特性

新兴的用户空间 NTB PMD （Poll-Mode Driver）[13, 25] 在不同服务器之间的 PCIe 域中提供了高速和可靠的 PCIe 连接。NTB PMD 的核心是通过支持单边 *Write* 和 *Read* 原语，在无需内核参与的情况下，允许用户级应用程序直接访问对端共享的远程内存。用户级 DPDK NTB 库进一步提供了 NTB 设备的配置和初始化、链路管理、硬件寄存器的写/读操作、内存分配、以及带有出入队列接口的 NTB 环形队列抽象。

NTB 的传输功能依赖于几个关键的硬件特性，包括消息和暂存寄存器（*Message/Scratchpad Registers*）、门铃寄存器（*Doorbell Register*）和内存窗口（*Memory Window*，*MW*）[141]。可读写的消息和暂存寄存器可以从互连的双边访问，并用于互连双边之间共享一些具有固定地址的控制消息。门铃寄存器会启用对等点之间的中断通知，将内存写入信号转换为对单独主机的中断。NTB 硬件实现了 *64* 比特的 *outbound/inbound* 门铃寄存器，其中每个比特都可以通过内存映射中断互连的对端主机。*MW* 使用基地址寄存器（*Base Address Register*，*BAR*）提供远端主机内存与本地内存空间之间的映射。每个 NTB 设备通过 *BAR* 管理一个或多个 *MW*。在 CPU *root complex* 初始枚举过程中，PCIe 内存空间会被分配给 *BAR*。

用户空间 NTB 通过基于共享 *MW* 的直接地址转译来实现内存映射 IO（Memory-Mapped I/O，MMIO）[165]。地址转译支持用于执行 *MW* 初始化的 *inbound/outbound* 转译接口。*Inbound* 转译在本地 NTB 端口上配置，而 *outbound* 转译由对等方配置。使用上述寄存器进行 *MW* 初始化的详细过程如下。

对于 *inbound* 转译：

（1） 本地 NTB 设备为每个共享 *MW* 分配了一个独立且连续的内存区域。

（2） 用分配的内存区域的转换地址初始化 NTB 配置中的每个 *MW*。

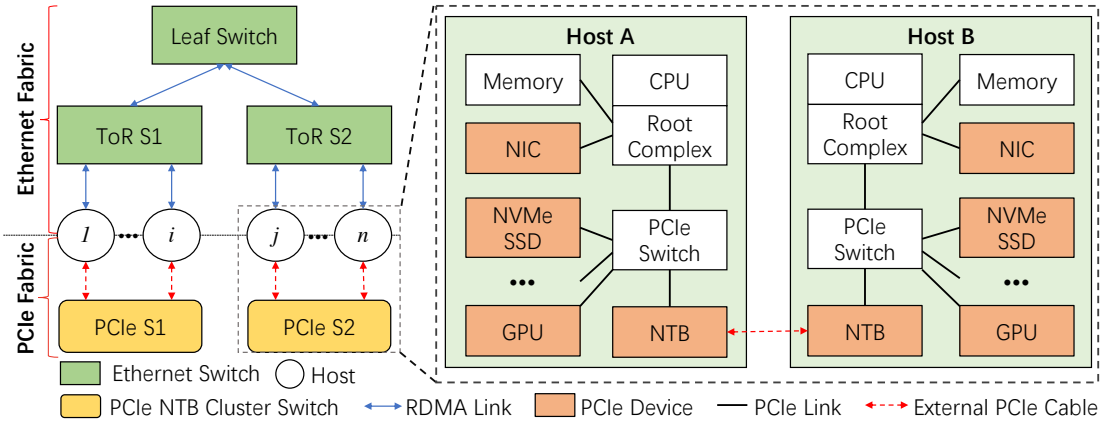


图 5-6　PCIe NTB 和 RDMA 混合部署的网络拓扑

（3）　利用暂存寄存器向对端发送上下文信息（例如，*MW* 的索引和固定大小的转译地址、*MW* 的计数和大小）。

（4）　对端的 NTB 设备将相应的 *outbound MW* 映射到转译后的地址，以便对端进程可以访问对应的共享内存区域（即 NTB 内存）。初始化完成后，双方使用门铃寄存器发回 *Peer-Up* 握手响应。

除了共享 *MW* 初始化由对端 NTB 设备完成之外，*outbound* 转译具有与 *inbound* 转译类似的过程。

我们以单边 NTB 远程 *Write* 的工作流为例[165, 166]。当用户级进程在共享 *MW* 的指定孔径上执行单边 *MMIO Write* 操作时，将发出 *PCIe Write* 事务并路由到指定的 NTB 设备。具体来说，NTB 设备将上述事务的地址转译为对端 PCIe 域中的新总线地址，并将事务转发给对端 NTB 设备。

当前的 DPDK NTB PMD 利用了几种性能优化技术[13, 140]。例如，Intel DPDK 减少了 PCIe 事务的数量，并通过禁用中断、采用写入合并（*Write Combining*，*WC*）[167] 和轮询主机内存中的 *Write Back* 描述符[72, 140] 来增加可用的 PCIe 带宽。用户空间 NTB PMD 还利用 Intel 数据直接访存技术（Data Direct I/O，DDIO）直接加速 NTB 设备和 CPU 缓存之间的数据移动，从而降低时延并增加带宽[168, 169]。

5.2.3　基于 PCIe NTB 的机架级网络通信

我们认为使用 PCIe 互连（PCIe Fabric）作为机架内高速网络技术是一个理想的选择，因为它不需要 PCIe 协议与网络协议之间的转译，从而带来更低的传输时延，网络拓扑如图 5-6 所示。但是，由于 PCIe NTB 的设计初衷是实现跨独立 PCIe 域之间的设备间通信（如跨主机之间的设备共享[23]），直接应用用户态原生 NTB 为机架规模的系统提供高速网络会存在如下三个重要挑战。

1.　挑战 1：通信抽象不匹配

原生 PCIe NTB 不友好兼容的主要原因是，原生 PCIe NTB 软件栈和应用程序所期望的编程抽象之间不匹配。应用程序开发者期望使用简单易用的高级网络抽象，从而聚焦在应

用业务逻辑的设计与实现。然而，原生 NTB 缺少通用的网络功能和面向连接的抽象（如网络套接字），仅暴露出硬件原语级别的编程抽象和面向全局的共享内存，缺少对共享内存状态的跟踪和维护，如图 5-4 所示。这使得开发者不得不投入大量时间仔细编写和调优与业务逻辑无关的硬件原语级代码，以紧耦合的形式贴合系统性能需求，才能发挥出原生 PCIe NTB 的高性能收益。例如，利用单边 NTB 远程写（*Remote Write*）原语在 NTB 映射的共享内存地址上完成客户端到服务端数据传输的完整流程如下：

（1） 两端均查询获取机器上的 PCIe NTB 设备；

（2） 两端通过 NTB 硬件寄存器启动跨主机的 NTB 设备之间的链路安装协商过程，交换检查 NTB 链路状态和上下文信息；

（3） 两端各自在 NTB 链路上下文信息中解析并获取由 NTB 映射的面向全局（并非面向连接的）的共享内存空间，包括本地和远端共享内存的虚拟地址和大小；

（4） 两端通过上述共享内存进行面向连接的握手，以确保本地和远程主机之间的共享内存状态准备就绪；

（5） 本地主机（即客户端）通过内存拷贝的方式，把待发送的数据从应用内存缓冲区中拷贝到 NTB 的远端共享内存上，而远端主机（即服务端）通过轮询一个共享内存标识字节来确定数据已完成传输以及数据大小，并基于数据进行业务处理；

（6） 两端通过共享内存进行显式的挥手以重置占用的共享内存状态；

（7） 两端通过 NTB 硬件寄存器协商关闭 NTB 设备，释放内存资源。

复杂的 NTB 传输过程使得应用程序优化很困难，需要系统开发人员认真细致地选择不同的 NTB 操作选项（如写原语和读原语的选择）和配置参数（如将 NTB 共享内存划分为多少个片段）。同时，作为通用的网络功能，基于软件的流量控制对于保证可靠的数据传输、避免拥塞甚至丢包至关重要。尽管最近的 NTB 环状内存设计[13, 170] 提出了简单的队列级的抽象以管理全局共享内存的状态，但并不是类似 RDMA 中的面向连接的发送和接收队列，距离应用程序所需的灵活的连接级抽象仍然很远。**在实现一个通用兼容的高级抽象的同时保留原生 NTB 的性能收益是很大的挑战**。

2. 挑战 2：缺少可扩展的 PCIe NTB 数据平面

原生 NTB 没有提供任何多核扩展性的数据平面机制来处理高并发流量负载。一方面，由于原始 NTB 驱动程序默认假设他们独占控制 NTB 设备[23]，原生用户态 NTB 栈仅支持以紧耦合的单进程独占方式来实现进程内的父子进程或线程之间的传输共享。这严重限制了 NTB 数据平面在多个应用进程间的扩展性，因为机架规模的系统通常需要多个应用程序进程的协同和共存。

另一方面，由于 NTB 硬件实现中地址转译的复杂性，映射出来的共享内存总大小通常是有限的。例如 Intel Xeon 处理器上每个 NTB 内存窗口仅仅支持 512MB[13] 大小的共享内存空间。这进一步限制了 NTB 数据平面的多核扩展性。首先，如果支持零拷贝，NTB 共享

内存的虚拟地址无法在多个应用进程之间共享。其次，如果支持进程之间的 NTB 传输共享，许多网络连接的内存系统需要的内存大小要远远超出当前 NTB 硬件支持的最大内存能力。例如，典型的键值存储 *memcached*[171] 依赖 64MB 的预分配内存区域和 1MB 的内存分配单元。因此，在多核机器上充分利用多核计算能力和有限的共享内存资源，从而在提高 NTB 传输扩展性的同时改进 CPU 使用效率，是一个重要挑战。

3. 挑战 3：缺少多租户或应用之间的性能隔离

原生 NTB 缺少一个中心化软件来从全局视角实现跨租户或应用之间的性能隔离，从而很难确保应用程序的服务质量。由于不同的应用程序有不同的流程模式，NTB 资源复用不可避免地引入多个租户或应用之间的性能干扰，即**线头阻塞问题**和**负载不均衡问题**。具体而言，在没有全局资源管理的情况下，一个应用进程对 NTB 共享内存资源的大量占用会直接影响其他应用进程的可用共享内存大小，从而影响系统的服务质量。假设以一个 NTB 共享内存队列作为一个资源复用单元，多个流（即应用进程）通过这个 NTB 共享内存队列传输数据。对于同一个复用单元内的所有网络流而言，会出现线头阻塞问题，即以小消息为主的时延敏感流会被以大消息为主的带宽敏感流阻塞，从而表现出极高的尾时延，很难满足严格的服务级目标。对于不同的复用单元而言，高并发网络流量下动态的流量负载（如随机断连、变化的流速率）会导致复用单元之间的负载倾斜问题，从而导致相对空闲的复用单元内的吞吐量偏低，而其他繁忙的复用单元内的吞吐量过于饱和。所以，**隔离不同流量模式的租户应用以实现高效公平的资源共享具有很大挑战**。

5.3 架构概览

NTSocks 的主要设计目标是解决上述三大挑战以提供兼容性、多核扩展性、性能隔离，同时预留出原生 NTB 的高性能。在本节中，我们会概述 NTSocks 的系统架构、线程模型，并给出一个基于 NTSocks 的数据传输案例。

5.3.1 设计目标与挑战

上一节提到的原生 PCIe NTB 的三个正交问题指向一个共同要求：即一个面向 PCIe NTB 互连支持全局资源管理的、统一、集中式的系统框架。将这个统一 NTB 传输框架转化为可实际应用的系统存在五个关键挑战：

- 如何设计支持全局 PCIe NTB 资源管理的集中式框架对应的系统架构？
- 系统如何在保留 NTB 性能优势的同时提供通用的网络功能抽象？
- 系统如何在 NTB 内存资源有限的情况下实现多核可扩展性？
- 系统数据平面如何保证连接级的传输可靠性？
- 系统如何隔离具有不同流量模式的各种上层应用程序？

5.3.2　NTSocks 架构

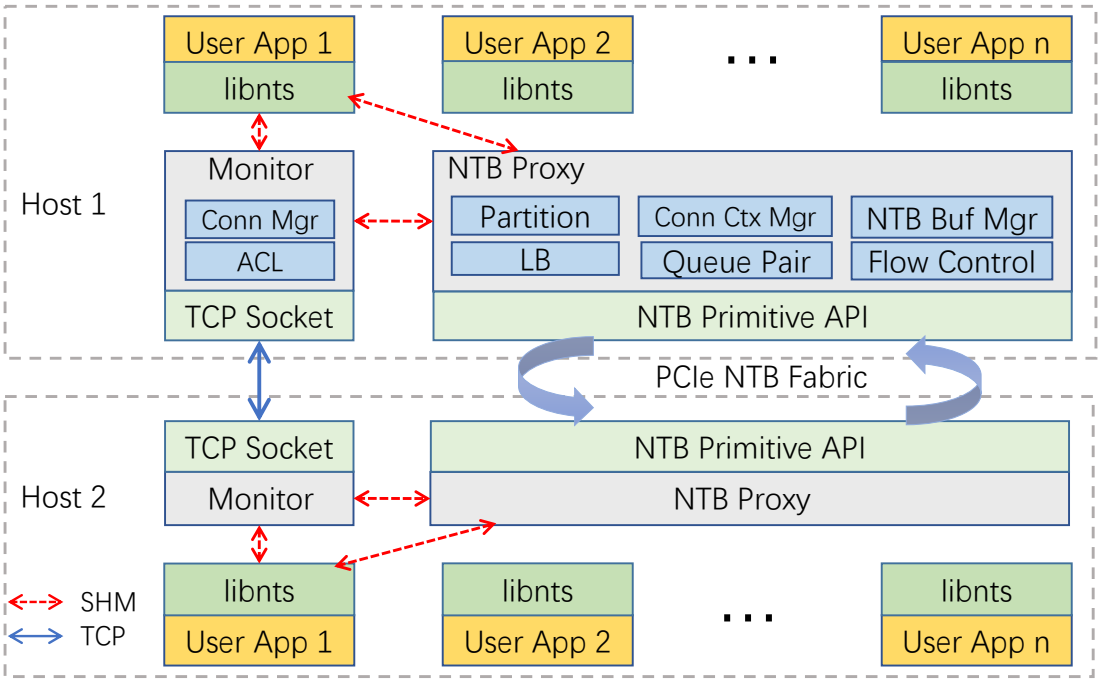


图 5-7　NTSocks 系统整体架构图

为了解决 PCIe NTB 互连应用于机架内高速网络的关键挑战，我们设计了 NTSocks，一个构建在 PCIe 互连上集中式的用户空间机架级的网络架构，为机架规模系统提供接口兼容、高性能、多核扩展且隔离的网络功能，允许多个应用高效的共享 NTB 传输资源。NTSocks 的核心思想是引入了一个**用户级间接层**，通过软硬件协同设计将原生低级的 NTB 栈转换成高级抽象，桥接了原生 NTB 和上层应用所需网络抽象之间的语义鸿沟。

图 5-7 展示了 NTSocks 的整体架构。NTSocks 有三个主要组件：用户级网络库 *libnts*，数据平面间接代理（*NTB Proxy*，*NTP*），和控制平面监控器（*NTB Monitor*，*NTM*）。这三个组件通过进程间共享内存（*SHared Memory*，*SHM*）通道互连协同。

Libnts 以运行时库的方式驻留于应用进程内，通过代码非侵入（即 *LD_PRELOAD* 的方式）或少量代码改动的方式为用户级应用程序提供 POSIX 兼容的通用网络套接字接口。它通过三种方法提供兼容且高性能的通信抽象：（1）一个连接级抽象，包含了一对每连接的服务于发送（*TX*）和接收（*RX*）的 SHM 无锁环形队列和内存池（第 5.4.1 节），（2）一个数据包模块，用于消息切片与合并来消除线头阻塞（第 5.4.3 节），和（3）一个 NTB 内存共享模型，用来优化应用缓存区、TX 与 RX 共享内存、NTB 映射的共享内存之间的内存拷贝，为不同的场景需求提供 *origin-nts*、*shm-nts*、*direct-nts* 三种不同的模式（第 5.4.5 节）。*Libnts* 协同 NTB 间接代理组件 *NTP* 实现了快速的数据路径操作（如 *read()* 和 *write()*）。

间接代理组件 ***NTP*** 作为数据平面以独立守护进程的形式运行在每个机器上，从全局视角管理 NTB 资源共享，为 *libnts* 驱动的应用程序提供 PCIe NTB 传输能力。*NTP* 通过多个关键模块来支持可扩展且隔离的 NTB 资源共享：（1）一个连接管理器（即 *Conn Ctx Mgr*），能够在控制平面 *NTM* 的协助下高效地缓存连接上下文信息如 TX/RX SHM 队列对（*Queue Pair*，QP），和数据包路由信息（第 5.4.1 节）；（2）一个 NTB 共享内存管理模块（即 *NTB*

Buf Mgr），将有限的 NTB 共享内存组织成无锁环形换冲区，并基于高性能的 NTB 原语提供高效的入队和出队操作（第 5.4.1 节）；（3）一个分区（*Partition*）抽象能够将多个 NTB 无锁内存缓冲区封装为多个并行的数据平面复用单元，每个复用单元（即一个分区）由 CPU 核心驱动并负责一组（具有同一目的地址的）连接的数据包转发，并提出一个核心-分区模型，能够按照具体的场景需求为每个分区分配 CPU 核心，以更好地权衡多核扩展性和 CPU 使用效率（第 5.4.2 节）；（4）一个负载均衡模块（*Load Balancing*，*LB*），实现了分区间连接分发调度，以消除负载倾斜，实现性能隔离（第 5.4.3 节）；和（5）一个接收端驱动的自适应流控（*Flow Control*，*FC*），实现可靠的面向连接的流量限速传输，避免软件层丢包（第 5.4.4 节）。*NTP* 通过控制应用程序进程和 *NTP* 之间的 TX-RX SHM 队列对来实施数据平面转发策略（如批量转发）。

软件控制器 *NTM* 也是以守护进程的形式运行在每个机器上，为所有加载 *libnts* 的应用程序提供数据平面的网络功能。它借助 TCP 套接字与对端的 *NTM* 交换 NTB 链路的元数据信息（如 NTB 链路状态、NTB 共享内存大小和分区数量），维护虚拟 IP 地址与 NTB 链路的映射表，连接建立（即握手）和断连（即挥手）。它在用户级全局地为每个连接分配虚拟的套接字 ID 和虚拟端口。它与 *libnts*、*NTP* 协同分别在握手（如 *connect()*、accept()）、挥手（如 *shutdown()*、*close()*）的时候创建或销毁连接级的上下文对象（如 *SHM QP*）。

5.3.3　线程模型

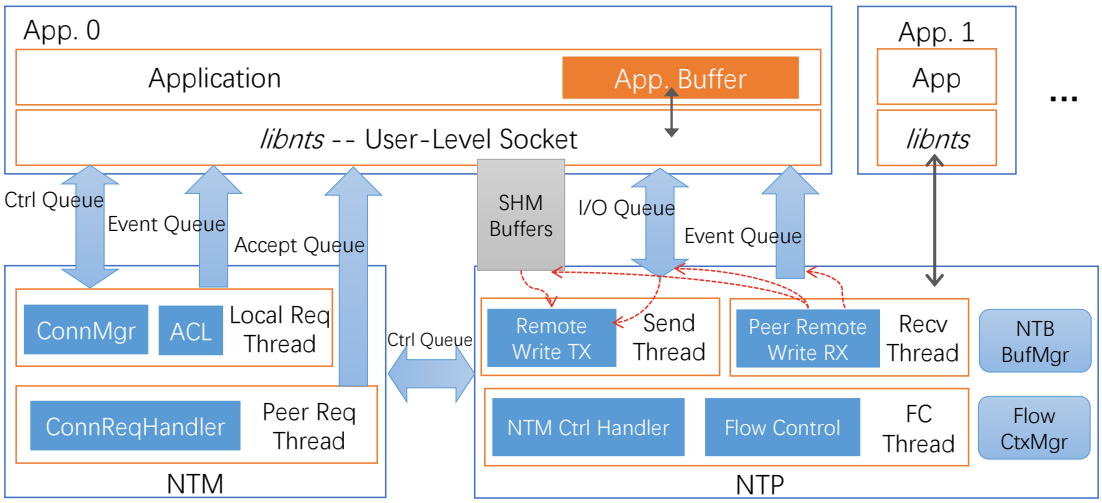


图 5-8　NTSocks 线程模型

图 5-8 展示了 *libnts*、*NTM* 和 *NTP* 之间交互的过程。多个重要角色的 SHM 队列和线程用来实现上述三个组件之间的高效协同。基于 *libnts* 的应用程序分别通过 *NTM* 和 *NTP* 实现控制平面和数据平面操作。

对于控制平面，*NTM* 利用本地请求线程（即 *Local Req Thread*）来处理控制队列（即 *Ctrl Queue*）中 *libnts* 的请求，例如套接字分配 *socket()*，端口绑定 *bind()*，*backlog* 初始化 *listen()*，主动握手 *connect()*，和套接字关闭回收 *close()*。*Local Req Thread* 还执行自定义的安全策略如访问控制列表（*Access Control List*，*ACL*）。NTM 中的对端请求线程（即 *Peer Req Thread*）主要负责监听远端连接请求（SYN），通过基于共享内存的控制队列指导 *NTP* 初始化对应的

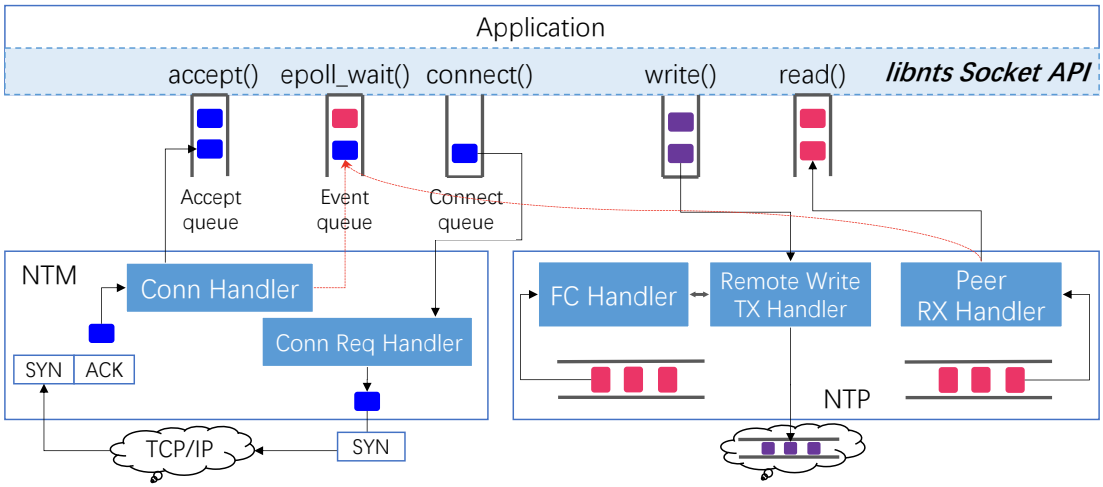


图 5-9　一个基于 NTSocks 的数据传输案例

连接上下文和 TX-RX SHM 队列对（即 *IO Queue*），进一步将请求建连的客户端套接字放入服务器套接字的接受队列（即 *Accept Queue*）最后预加载 *libnts* 的应用程序调用 *accept()*，从 *Accept* 队列中取出建连成功的套接字实例。

对于数据平面，*NTP* 包含三个主要线程：发送线程（即 *TX* 线程）、接收线程（即 *RX* 线程）和 *FC* 流量控制线程。对于数据发送，应用通过调用 *libnts* 中的 *write()* 将应用缓存区中的数据放入网络套接字对应的发送 I/O 队列（*TX IO Queue*）中。发送线程以 *Round-Robin* 的方式轮询所有连接，从每个连接对应的发送 I/O 队列中取出消息数据包，增加路由元数据到数据包头中，然后利用 NTB 内存操作原语将序列化的数据包放入到远端 NTB 环形缓冲区中。如果 *POLLOUT* 事件被该连接标记，一个可写（*writable*）事件将会生成并放入对应的事件队列（*Event Queue*）中，并通知对应的套接字可以继续发送数据。对于数据接收，接收线程会以轮询的方式从本地的 NTB 环形缓冲区中取出数据包，通过反序列化解析查询对应的连接上下文，并把数据包放入该连接对应的接收 I/O 队列（*RX IO Queue*）。如果 *POLLIN* 事件被连接标记，一个可读（*readable*）事件会生成并放入到对应的事件队列中，以通知 *libnts* 应用调用 *read()* 接收数据。流量控制线程还通过一个专门的 NTB 控制环形缓冲区提供可靠的面向连接的流量控制功能。图 5-9 详细地展示了上述数据传输过程。

5.4　NTSocks 系统设计

在上述 NTSocks 的架构基础之上，本节聚焦在数据路径（即 *libnts* 和 *NTP*）上的关键设计点，从而合理地权衡多个面向轻量级协议栈的设计目标。

5.4.1　通用的通信抽象

为了解决不匹配的通信抽象问题，我们旨在设计一个通用目的的编程抽象（即网络套接字）的同时，保留接近裸 NTB 的高性能。换句话说，即实现**友好的兼容性**和**高性能**之间的最佳权衡。为此，我们结合了多种数据路径优化机制来设计了一个灵活的兼容 POSIX 套接字且高性能的非侵入式编程抽象（如图 5-9 所示）。

1. 连接级抽象

NTSocks 以 *LD_PRELOAD* 预加载的方式通过 *libnts* 库拦截每个 Linux 网络套接字相关的函数调用，并通过 *NTP* 转发到 NTB 硬件。因此，在 *libnts* 运行时库和 *NTP* 间接代理之间，NTSocks 提供一个高效的基于共享内存 SHM 的连接级抽象，这是保证兼容性和原生 NTB 性能收益的一个基础环节。本小节描述了连接级抽象（即每连接的 *TX-RX SHM* 队列对和内存池）的无锁设计。

无锁化每连接发送、接收 SHM 队列对。当建立一个连接时，一个唯一对应的 *TX-RX SHM* 队列对会在 *NTP* 和 *libnts* 之间建立。因为避免数据路径上锁的使用是一个关键的设计原则，在上述 SHM 队列上的基本操作（如 *push()* 和 *pop()*）均以无锁的方式实现。对于吞吐敏感的场景，我们还设计了无锁批量处理操作（如 *bulk_push()* 和 *bulk_pop()*）来减少 SHM 队列上原子操作的数量，以提高每秒 IO 数量。用户可以根据应用需求自定义该批量大小参数。

每连接发送、接收 SHM 池。在 SHM IO 队列的朴素实现中，入队时需要从输入缓冲区到 SHM 缓冲区的数据拷贝，出队时需要从 SHM 缓冲区到输出缓冲区的数据拷贝。这带来了不可容忍的性能降低，因为数据路径 *libnts* 和 *NTP* 之间的 *push()*、*pop()* 操作会频繁发生，导致高昂的系统调用和内存拷贝开销。为此，我们提出了服务于每个连接的无锁 *TX-RX* 共享内存池来移除冗余的数据拷贝。具体而言，由于 *libnts* 应用进程和 *NTP* 守护进程之间的虚拟地址不同，在进程间共享内存池中我们使用内存偏移量作为索引。在入队的时候，从共享内存池中分配内存作为输入缓冲区，仅仅将对应的偏移量索引推入共享内存队列中。在出队的时候，利用弹出的偏移量索引映射得到对应的共享内存缓冲区，即输出缓冲区。在完成数据转发过程后，已分配的共享内存缓冲区会被共享内存池回收再利用。

2. 共享的无锁 NTB 环形缓冲区

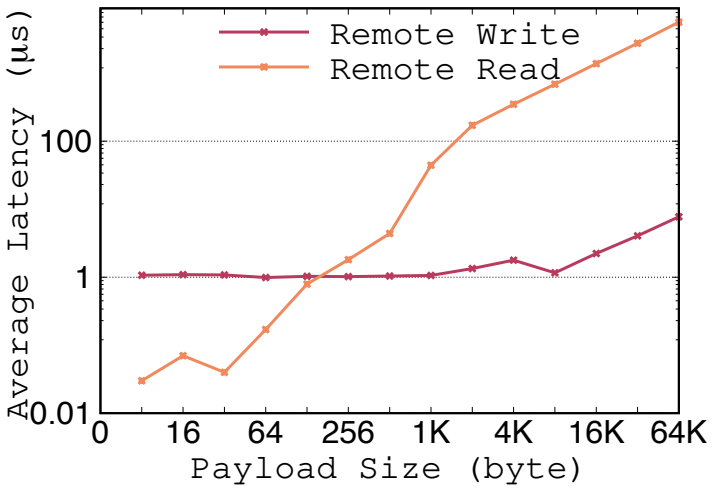


图 5-10　PCIe NTB Write/Read 原语的往返时延（RTT）对比

NTB 单边远程写比读原语有更好的性能。为了彻底地理解不同 NTB 原语（即单边远程写、读原语）的特性，我们通过端到端的数据收发实验系统性地评估了在 NTB 远程共享内

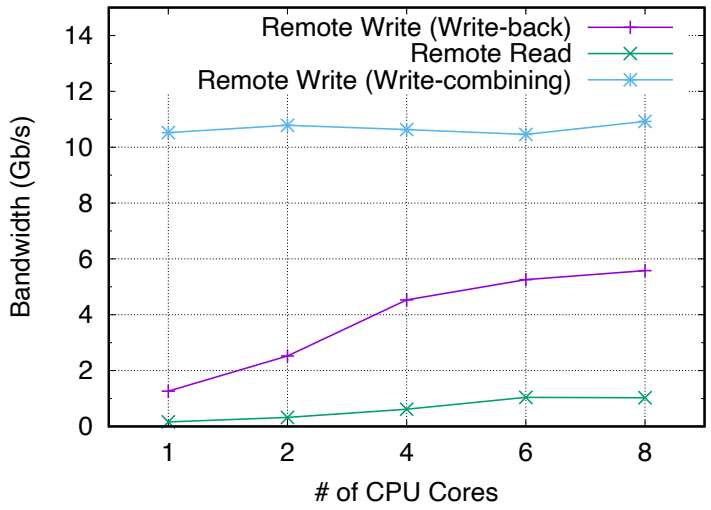


图 5-11　PCIe NTB Write/Read 原语在 64 字节消息下的吞吐量对比

存上不同原语操作的往返时延（Round-Trip Time, RTT）和消息吞吐，如图 5-10 和 5-11 所示。我们发现：

- 远程写操作（*write-combining* 模式）比远程读操作在多数情况下有高达 2 个数量级更低且更稳定的往返时延。如图 5-10 所示，在时延方面，对于小于等于 256 字节的消息，远程写操作比远程读操作有 25% 至 3 倍略高的时延。这是因为 *Write-Combining*（*WC*）定义了一个内部的 *WC* 缓冲区并用来组合多个小消息块（即较小的事务层数据包）以减少内存访问提高吞吐，而对映射的 *WC* 内存（此处即 NTB 共享内存空间）的写操作不会缓存在 CPU L1 和 L2 缓存中，也不会被 PCIe 事务层立即处理转发，而是直接到 *WC* 缓冲区被滞留延迟，以将多个很小的内存写事务层数据包（Transaction Layer Packet，TLP）合并成一个内存写 TLP。因此对于小消息（小于 256 字节）而言，写操作有较高的传输时延，尽管这个时延开销是可接受的。而对于大于 256 字节的消息，远程写操作比远程读操作有高达两个数量级的更低时延。这是因为：1）*WC* 对于大消息传输不会触发事务层数据包合并，从而立即被 PCIe 事务层处理转发；2）从事务层数据包处理的角度，相比内存写事务层数据包，内存读操作在生成一个内存读事务层数据包之后，需要等待一个来自目的端事务层已确认的完成事务层数据包，该完成事务层数据包内携带了内存读操作请求的数据，以 TLP 数据包单元大小为 128 字节为例，大消息通常需要在 PCIe 事务层切分为数量更多的 TLP 数据包，这进一步加剧了远程读和远程写在时延上的差距。

- 远程写操作（*write-combining* 模式）比远程读操作实现 10 倍以上更高的（64 字节）消息吞吐量。如图 5-11 所示，在吞吐方面，*WC* 写操作在 64 字节消息下的吞吐收益，主要是由于：1）写合并以批量处理的形式减少了内存访问操作；2）在 PCIe 事务层协议中，内存读操作比内存写操作多了一次额外的来自对端的确认事务层数据包传输和处理。

- PCIe 远程内存写操作有 *write-combining* 和 *write-back* 两种模式，*write-combining* 比 *write-back* 模式有 2 到 10 倍更高的吞吐，如图 5-11 所示。这是因为相比 *write-back* 模式，PCIe *write-combining* 模式在硬件允许的情况下将多个 PCIe 事务层数据包合并成一个 PCIe 事务，通过内存映射 IO（Memory-Mapped IO，MMIO）写到 PCIe NTB 硬件中，从而降低

了 PCIe 内存写事务的数量，提高了吞吐。

基于单边写原语的 NTB 环形缓冲区。 NTB 原语的特性发现促使我们利用 WC 模式下的单边远程写和本地读来设计在 NTB 共享内存上的无锁环形缓冲区，以此达到 *NTP* 数据转发的理想超低时延和高吞吐。

我们将有限的 NTB 内存（代指 NTB 映射的共享内存）组织成一个包含 64 字节写索引（*write_index*）和读索引（*read_index*）的无锁环形缓冲区。其中，单边远程写被用于执行类似 *push* 的入队发送操作，而本地读被用于执行类似 *pop* 的出队接收操作。对于本地主机在远程 NTB 环形缓冲区上的数据发送操作，*NTP* 中的发送工作线程（即 *TX worker thread*）首先将消息从每连接对应的发送共享内存队列中取出，并利用远程写原语转发到远端 NTB 内存缓冲区中，然后更新本地缓存的 *write_index*。对于本地主机在本地 NTB 环形缓冲区上的数据接收操作，*NTP* 内的接收工作线程（即 *RX worker thread*）首先轮询当前 *read_index* 指向元素位置的头部元数据消息大小 *msg_len* 字段，如果 *msg_len* 大于 0，则说明有新的消息到达，然后解析该消息得到对应的连接上下文，并转发该消息到对应的每连接接收共享内存队列中，最后更新本地的 *read_index*。

为了判断远程的 NTB 环形缓冲区是否已满，发送端利用一个本地的影子读索引（即 *shadow read_index*）来跟踪远端的 *read_index*。具体而言，在发送端每次执行消息发送之前，它首先利用本地的下一个 *write_index* 和影子 *read_index* 来估计远端 NTB 环形队列中待处理消息的占比 *pending_rate*。一旦 *pending_rate* 大于 *0.5*，发送端立即标记下一个消息的元数据控制字段来请求远程 *read_index* 的同步。对于接收端，一旦在接收的消息中发现上述标记，它会主动地通过单边写原语将本地最新的 *read_index* 同步到源端的影子 *read_index*。

为了保证每个消息在接收时数据内容和元数据的一致性，*NTP* 中的发送工作线程会在每个消息数据包元数据写入之前写入数据内容。这是因为我们设定每个消息的元数据是 8 个字节，NTB 远程写原语背后的 8 字节 *MOV* 指令是原子操作。

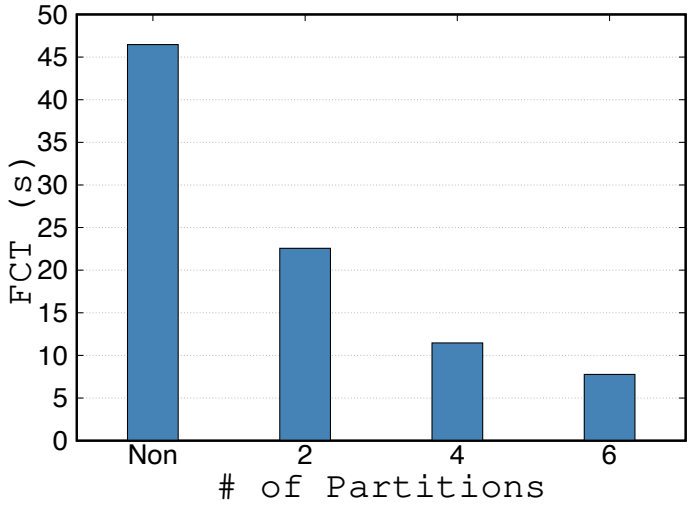


图 5-12　在并发流量负载场景下，不同分区数量对应的 NTSocks 流完成时间（Flow Completion Time, FCT）

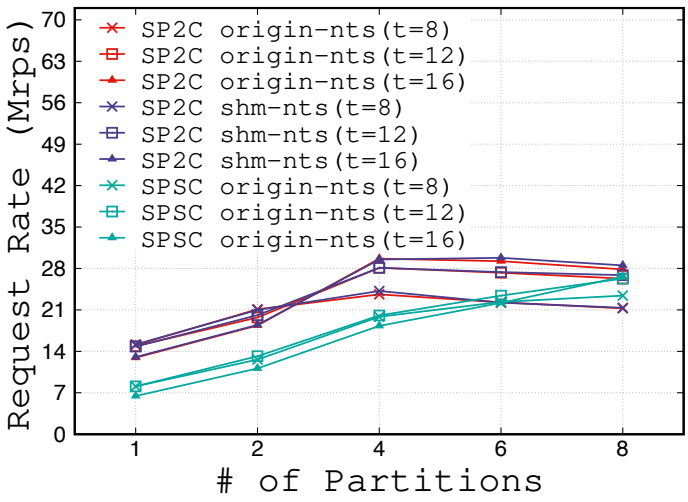


图 5-13　在并发流量负载场景下，不同分区数量对应的 NTSocks 小消息请求率

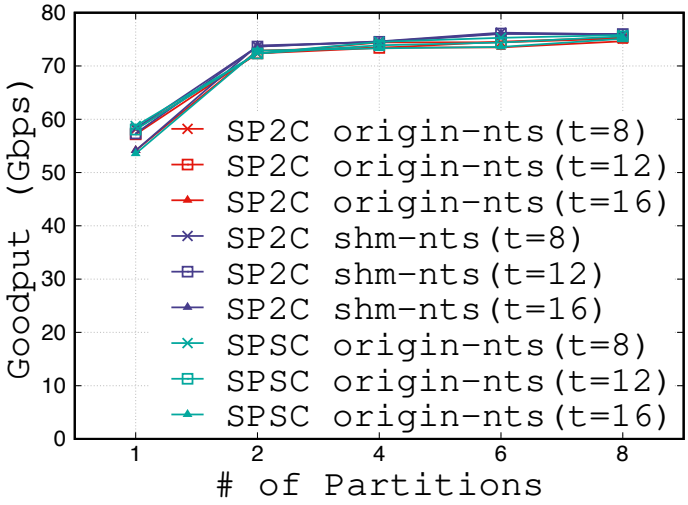


图 5-14　在并发流量负载场景下，不同分区数量对应的 NTSocks 吞吐量

5.4.2　Partition 分区抽象

由于有限的 NTB 共享内存空间和登纳德缩放比例定律（Dennard Scaling）的终结，实现可扩展的 NTB 资源共享至关重要。然而，NTSocks 数据平面的初始设计仅将整个 NTB 共享内存空间组织称全局唯一的共享环形缓冲区，这种设计无法释放多核机器上多核计算的潜力，在突发流量负载下数据面是潜在的性能瓶颈。我们在 PCIe NTB 互连的两台机器之间设计了一个扩展性验证实验：发送端并发地发送 12 条固定大小的压力流到接收端，我们基于统计测量的所有流完成时间（Flow Completion Time，FCT）计算出平均流完成时间。图 5-12 显示原生数据平面的朴素设计很难提供多核可扩展的性能。

为了解锁数据平面的多核扩展的性能，NTSocks 进一步提出了分区 *Partition* 抽象。分区的核心思想是将有限的 NTB 共享内存空间分割成多个并行的复用单元，每个单元是由 CPU 核驱动的（如绑定专用的 TX 和 RX 核）并由一组连接复用。图 5-12 中的结果显示随着分区数量的增加，平均流完成时间接近线性地降低，并在分区数为 6 时收敛至平稳状态。图 5-13

和 5-14 中的结果还显示 NTSocks 通过多核驱动的分区抽象可以很容易饱和 PCIe NTB 带宽，在并发压力流量下实现更好的多核性能缩放。

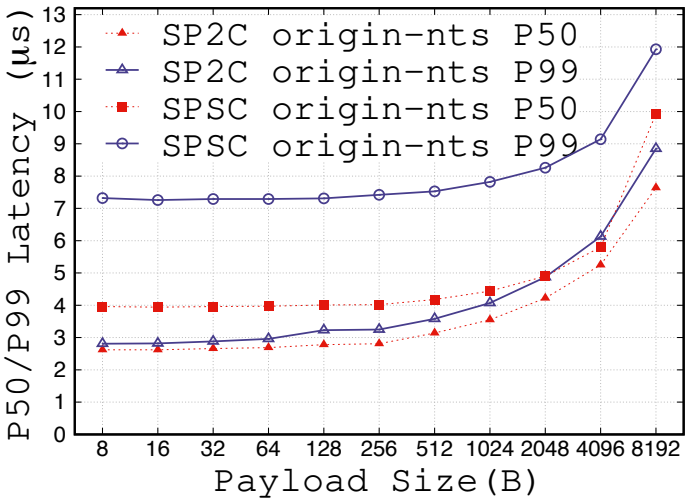


图 5-15　NTSocks 在 SP2C 和 SPSC 两种分区模式下的时延对比

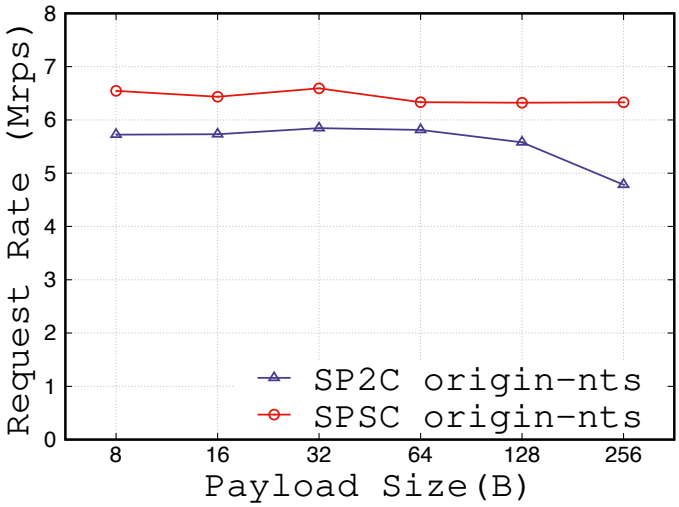


图 5-16　NTSocks 在 SP2C 和 SPSC 两种分区模式下的小消息请求率对比

改进 CPU 使用效率。上述分区设计需要使用两个专用内核分别作为 TX/RX worker，称为单分区 2 核（*Single Partition with 2 Cores*，*SP2C*）模式。虽然 SP2C 模式友好地确保了在时延敏感的小消息情况下超低的时延（*<5us*）和尾时延，但它带来了很高的 CPU 开销。因此，我们进一步提出了单分区单核（*Single Partition with Single Core*，*SPSC*）。SPSC 模式中每个分区只需要一个专用 CPU 核心来交替进行一组连接的数据发送、接收操作。我们评估了两种模式在并发压力流下的性能，图 5-15 中的时延对比展示了 *SPSC* 模式相比于 *SP2C* 在中位时延和 99% 尾时延引入了可接受的时延开销，这是由于 *SPSC* 模式下的数据面工作线程需要在数据包发送和接收两种任务之间进行切换。图 5-17 中带宽实验结果表明 *SPSC* 模式也能随着消息大小的增加而饱和 PCIe NTB 带宽。但相较于 *SP2C*，*SPSC* 实现的小消息请求率差了 1 倍，这是由单个 CPU 核在数据发送和接收任务之间切换产生的开销导致的。在

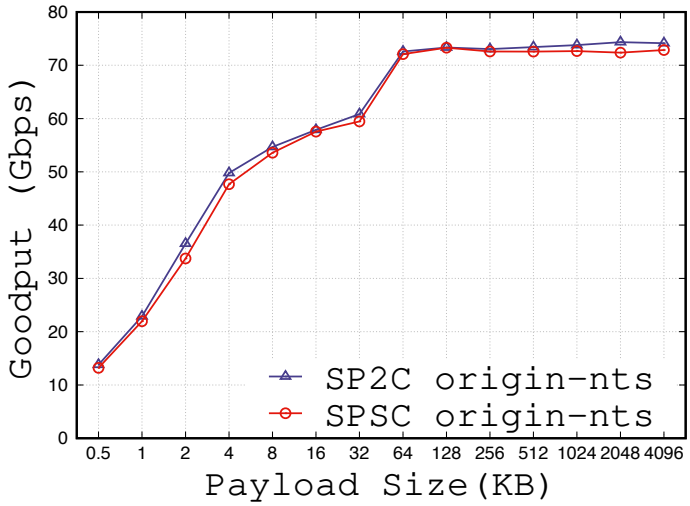


图 5-17　NTSocks 在 SP2C 和 SPSC 两种分区模式下的总带宽对比

部署使用 NTSocks 系统时，用户可以为时延敏感的场景选择 *SP2C*（图 5-16），而对带宽敏感的应用场景选择 *SPSC* 模式。在没有特别声明的情况下，本文默认使用 *SP2C* 模式。

5.4.3　性能隔离

接下来，为了跨应用程序的性能隔离，我们利用消息切片和合并设计保证分区内多个连接之间资源共享的公平性，同时利用 *Round-Robin* 连接粒度的调度策略来实现多个分区间的负载平衡。

分区内资源复用的公平性。一组连接共享相同的分区和相应的发送、接收工作线程，这意味着以小消息为主导的时延敏感的流可能会被以大消息为主导的带宽敏感流阻塞。为此，发送工作线程需要在每轮轮询期间，以有限的批处理大小从每个连接的发送共享内存队列中批量地转发消息。同时，消息切片和合并对于将大消息拆分为一组固定大小（即分片大小 *mtu size*）的数据包是必不可少的，这可以实现一个分区内混合小消息流和大消息流之间的公平性。问题是，NTSocks 在什么位置执行消息切片、合并操作更合适呢？为了避免消息切片、合并操作影响 *NTP* 进程的数据发送、接收效率，我们将对应逻辑实现在应用进程内的 *libnts* 运行时库中。

分区之间的负载均衡。由于 NTSocks 中有多个分区，我们应该将传入的新连接分配到哪个分区呢？分区之间的负载倾斜会影响上层应用的整体服务质量（Quality of Service，QoS），并降低 PCIe NTB 传输的效率。与分区内消息切片和合并密切协调，Round-Robin 连接分发机制可以有效地平衡分区间负载。

5.4.4　接收端驱动的流量控制

为了使端到端的连接级传输更加可靠，NTSocks 采用了接收器驱动的自适应流量控制机制。这旨在避免每个连接的接收 SHM 队列溢出。虽然 PCIe NTB 互连是无损的，但我们不能保证连接级共享内存接收队列是不丢包的。接收端 NTP 在接收、分发数据包时，一旦

接收共享内存队列的可用长度触发了拥塞阈值，就会产生拥塞信号，并通过一个基于 NTB 内存的专用控制环缓冲区从接收端传输到发送端。然后发送方可以根据当前的拥塞程度，自适应地调整对端发送共享内存队列上的发送速率。

5.4.5　零拷贝支持

从高层次的角度来看，非侵入式 libnts 库（即 *origin-nts*）使用的共享内存缓冲区充当应用程序缓冲区和 NTB 内存缓冲区之间的间接层。为了满足各种场景的需求，我们通过优化应用缓冲区、共享内存缓冲区和 NTB 缓冲区之间的内存拷贝，减少内存访问和拷贝次数，实现了 *origin-nts*、*shm-nts* 和 *direct-nts* 三种模式。特别是 *origin-nts* 实现了非侵入式接口，*direct-nts* 以代码侵入的方式提供了关键数据路径上零拷贝支持，*shm-nts* 是二者的折衷设计，具体的数据传输运行流程如图 5-18 所示。

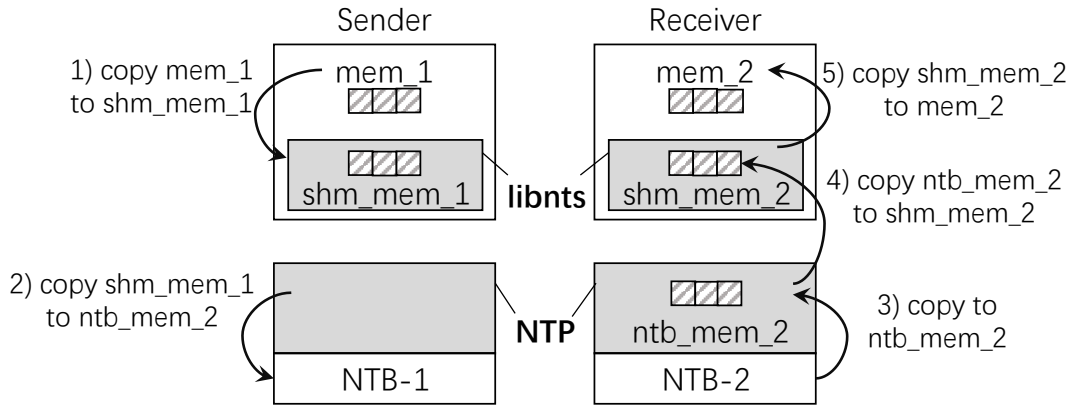
零拷贝： NTSocks 零拷贝的核心思想是通过 VFIO（Virtual Function I/O）驱动将远端 NTB 内存缓冲区透明地暴露给本地应用进程。允许本地应用访问对端 NTB 空间需要两个步骤：首先，应用进程需要获取本地 NTB 设备的 VFIO 文件描述符（File Descriptor，FD）。其次，应用程序需要根据该 VFIO FD 使用内存映射技术（*mmap()*）来映射 NTB 设备对应的虚拟内存地址空间。内存映射的偏移量和大小由获取得到的本地 NTB 设备的 *vfio_region_info* 结构体对象提供。NTP 可以通过 Unix 套接字直接将获取到的 VFIO FD、大小和偏移量同步给应用进程，然后应用进程根据上述参数，利用内存映射技术把 NTB 对应的物理内存重新映射到该进程内的虚拟地址空间。如此，应用进程对映射得到的虚拟地址的内存操作会通过背后的内存映射 IO（Memory Mapped IO，MMIO）同步到远端 NTB 共享内存中。

5.5　NTSocks 系统实现

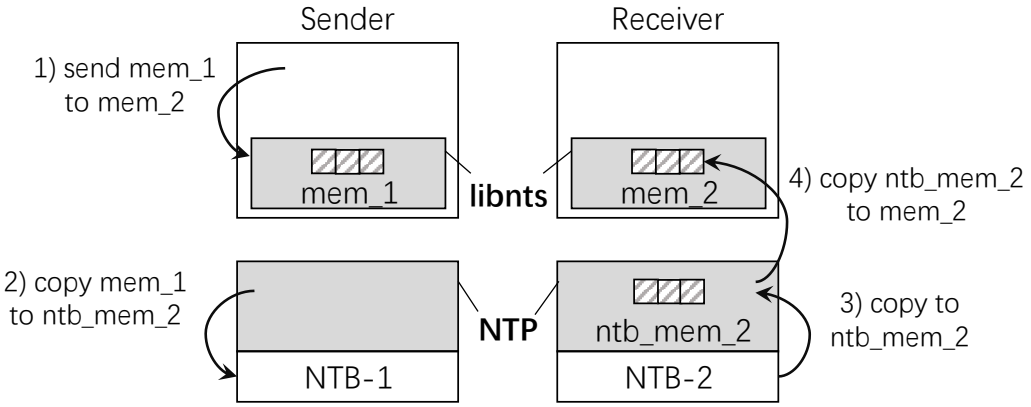
NTSocks 的实现包含了 *NTP*、*libnts*、和 *NTM* 三大组件。NTSocks 主要是使用 C 语言实现。基于用户态 DPDK NTB 轮询驱动程序，我们用大约 2500 行 C 语言代码从头实现了关键数据平面组件 *NTP*。运行时库 *libnts* 可以直接链接到应用程序，用于提供所有 POSIX 套接字相关的函数调用，它的逻辑实现用了约 3700 行 C 语言代码。控制平面 *NTM* 以守护进程的方式运行在每个机器上，存储了用户定义的控制平面信息，提供 IP 端口分配、访问控制、故障迁移、TCP 回退的能力，它的实现用了约 4100 行 C 语言代码。上述三个组件均依赖于一个约 4000 行 C 语言代码实现的定制化通用功能库 *libnts-utils*，该库封装了进程间 SHM 通信协议、共享内存池、哈希函数等通用功能。该实现目前支持 64 位 X86 架构，适配到其他平台不需要很多代码改动。整个 NTSocks 系统以用户态进程的形式运行在没有任何改动的 Linux 系统环境中。

控制平面和数据平面策略： NTSocks 可以处理来自应用进程的控制平面和数据平面操作，因而可以支持对应的控制和数据平面策略，包括流优先级、资源分区数、每分区维护的最大连接数、每分区带宽限制等。

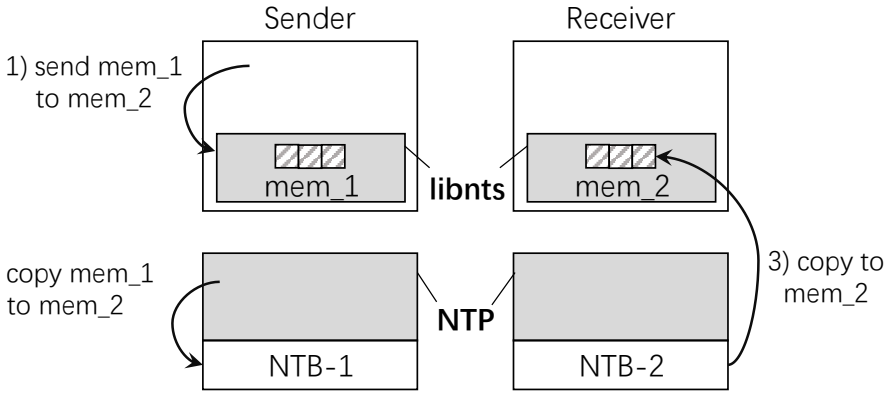
以控制平面策略为例，NTSocks 下放执行每个分区上可以支持的最大连接数配额，因为每个分区上维护的连接数量过多会引入不可避免的排队等待时延。这个控制平面策略避免



(a) *origin-nts* 的工作流程



(b) *shm-nts* 的工作流程



(c) *direct-nts* 的工作流程

图 5-18　当 Sender 发送 *mem_1* 到 Receiver 的 *mem_2* 时，NTSocks 三种模式在数据路径上的运行流程对比

了应用进程或租户在一个分区上创建太多的连接，从而影响了同一复用单元上的其他应用进程或租户。用户可以根据服务级目标的时延要求配置这个控制平面策略。

以数据平面策略为例，NTSocks 在零拷贝模式 *direct-nts* 下需要通过统一的内存分配器接口严格控制每个应用进程可用的 NTB 共享内存的配额，这是因为零拷贝将 NTB 映射内存透明化地暴露给了上层应用程序，一个应用进程占用了大量的 NTB 内存必然会影响其他应用的可用 NTB 内存，从而影响系统服务质量。

流优先级与服务质量（Quality of Service，QoS）： 在数据平面 QoS 策略实现中，NTSocks

默认为网络流定义了 8 个优先级，0 代表优先级最高，7 代表优先级最低，并且优先级数量是可配置的。对于数据平面 *NTP* 组件，我们定义了 QoS 单元，即一个优先级对应一个分区，每个分区大小一致。一个 QoS 单元内由一个 CPU 核心驱动 8 个优先级对应的 8 个分区，这个 CPU 核优先轮询转发高优先级分区内的所有连接，如果一次轮询中分区没有数据转发，则依次轮询下一个优先级分区。为了避免低优先级分区数据包被高优先级分区的持续数据流饥饿式地阻塞，我们设定了一个 CPU 核在每个优先级分区上的最大 Round-Robin 循环轮询次数阈值，即使高优先级分区内有待转发数据，CPU 核心也会切换到下一个优先级分区中进行数据转发。对于上层每个连接，我们在 *libnts* 库中提供专门设置每连接优先级的接口（即 *nts_conn_set_prior(int socket_fd, int prior)*），应用程序可以显式地设置一个新连接对应的优先级，NTSocks 会在 *NTM* 的配合下将该连接分配到对应优先级的 QoS 分区中。为了保证不同场景下的服务级目标，这需要用户根据场景需求来协同优化配置每分区最大连接数配额、每优先级分区最大循环轮询次数阈值。为了提高横向扩展能力，用户可以在系统启动时指定多个 CPU 核驱动的多个 QoS 单元。

数据包批量转发：在关键数据路径上，在每连接共享内存队列对上的每次出队或入队操作需要一次原子操作以保证事务一致性，NTSocks 支持在该队列上的数据包批量转发以提高消息吞吐量，N 个数据包的 1 次批量入队或出队可以将队列上的原子操作从 N 降低到 1。在数据发送方向上，*libnts* 中的 *write()* 函数调用会将切片后的数据包以指定批量大小的方式入队到对应的发送共享内存队列中，而 *NTP* 中的分区 CPU 核在轮询到该连接 TX SHM 队列时，则会批量出队，并转发到 NTB 环形缓冲区中。在数据接收方向上也是类似过程。

写合并（*write-combining*）与写回（*write-back*）：尽管 NTSocks 默认对 PCIe NTB 映射的共享内存空间使用写合并 *WC* 模式来折中小消息时延和吞吐，它仍然支持在 Linux 系统的*/proc/mtrr* 内核配置文件中，显式地将 PCIe NTB 硬件对应的物理内存基地址属性设置为 *write-back* 类型，从而支持写回 *write-back* 模式来得到更好的小消息时延。

主机内应用进程间通信：传统的主机内应用进程之间的通信通常基于以太网卡的环回（*Loopback*）流量传输，在主机内高并发流量负载情况下会大量占用本地主机上的 PCIe 总线带宽，从而干扰降低基于 PCIe NTB 的主机间通信的传输性能。为了彻底消除上述干扰所带来的性能降低，NTSocks 利用进程间共享内存通道来取代以太网卡环回流量。

5.6　应用与性能评估

本节主要评估了 *NTSocks* 系统的性能和开销。我们首先展示了 NTSocks 不仅可以在微基准测试实验中实现极佳的通信性能（第 5.6.2、5.6.3、5.6.4 节），还可以大幅加速生产环境中的实际应用系统（第 5.6.5 节）。我们旨在回答以下几个问题：

1. 在不同的工作负载下，NTSocks 相比其他网络协议栈的套接字系统能达到多好的时延和吞吐收益？NTSocks 的内部机制对于观察到的性能有怎样的影响？（第 5.6.2.1 节）

2. 相比其他典型网络协议栈，NTSocks 是否能够实现最优的多核性能缩放和性能隔离？（第 5.6.3 节）

3. 在机架内外 PCIe NTB 和 RDMA 混合部署的情况下，将原来机架内的 RDMA 流量卸载到 NTSocks，是否能够在缓解 RDMA Incast 拥塞的同时提供高性能？（第 5.6.4 节）

4. NTSocks 所带来的通信收益能否为实际应用系统带来端到端的性能提升？（第 5.6.5 节）

需要注意的是，由于生产环境中实际应用系统会重叠逻辑计算处理和数据传输，端到端性能的小幅度提升意味着在网络通信上的巨大改进。

5.6.1 实验环境与方法

实验安装：我们的测试床主要是运行在 PCIe NTB 和 RoCE 互连网络环境中。每个服务器装有 2 个 CPU 插槽，32 个 Intel Xeon Gold 5218 CPU 物理核心，64GB 内存，PCIe GEN 3×16 总线，一个 80Gbps NTB 芯片设计规格的非透明桥适配器（已向 Intel 厂商确认当前 NTB 实验平台的芯片设计规格为 80Gbps），一个 Mellanox ConnectX-5 网卡。RoCE 网络利用 Mellanox SN2100 交换机作为机顶交换机。运行的操作系统是 Ubuntu 18.04.5，Linux 内核版本是 5.1.3。实验环境中所用到的 PCIe NTB 适配器均为 Intel 生产的实验平台性质的样板卡。由于 PCIe NTB 硬件数量的限制，测试床中仅有两台服务器配有 PCIe NTB 适配器。由于缺少 PCIe 交换机，两台服务器通过 PCIe NTB 背连（back-to-back）方式互连，由于 PCIe NTB 是点到点通信，数据平面上的一跳中继 PCIe 交换机的时延开销可以忽略不计[23]，因此基于 PCIe 交换机的一跳 PCIe 拓扑和背连方式时延开销相差无几。
评估的系统：我们主要比较 NTSocks 与裸 NTB（Raw NTB）、完美的 RDMA 套接字（如 *LibVMA*[144]）的性能，从而说明 NTSocks 能够达到最优的性能。我们将会展示 NTSocks 能够以最小的性能损失代价，为上层多个应用进程提供虚拟化的 PCIe NTB 网络。我们还将 NTSocks 与广泛部署的 Linux 内核套接字（Linux Socket）和用户态网络协议栈 VPP[74] 进行性能对比。我们将 NTSocks 在不同模式下（即 *origin-nts*、*shm-nts*、*direct-nts*）的性能进行了对比分析，以展示 NTSocks 可以满足不同场景的性能需求。除非另有说明，否则 NTSocks 默认采用单分区双核 *SP2C* 模式。
实际应用：为了展示 NTSocks 的兼容性、在实际应用系统中的性能收益，我们以没有代码改动或最小的代码改动将代表性的键值存储系统（*Key-Value Store*，*KVS*）、Nginx Web 服务器、Apache 基准测试工具（*Apache benchmark tool*，*ab*）移植到了 NTSocks 上，并在不同的工作负载下比较 NTSocks 与 Linux 套接字、RDMA 套接字的应用性能。

- **键值存储系统：**是一个典型的以小消息为主、时延敏感的应用场景，我们基于当前流行的 *MurmurHash* 算法构建了一个基于 POSIX 套接字的通用键值存储系统，支持高性能的键值插入、删除、更新和查询操作。我们用 Linux 系统环境变量 *LD_PRELOAD* 分别将 *libnts* 和 *libvma* 以动态链接库的形式，实现了针对 KVS 的代码非侵入式 NTSocks 支持（称为 *nts-kv*）和 RDMA 套接字（称为 *RDMA-kv*）支持。我们使用了广为人知的 YCSB 提供各种不同的键值操作工作负载，来评测 *nts-kv*、*RDMA-kv*、*TCP Redis* 键值存储系统的应用性能。

- **Nginx：**是一个广泛应用的事件驱动的 HTTP 反向代理服务器，是一个以大消息主导、带宽敏感的应用场景。我们使用 *direct-nts* 提供的零拷贝套接字和事件接口来替

代 Nginx 的对应接口，以实现 Web 服务器在数据路径上的零拷贝低开销，这仅需要约 150 行 C 语言的代码改动来构建基于 NTSocks 的 Nginx，也说明了 NTSocks 零拷贝接口的易用性和灵活性。

- **Apache 基准测试工具：**是一个 HTTP 服务器的性能评测工具，能够生成不同的 HTTP 请求负载[172]。它采用单线程 IO 模型，作为一个 HTTP 客户端用来度量一个 Web 服务器的性能。原生 *ab* 依赖于可移植的运行时库 APR（Apache Portable Runtime）提供网络接口，APR 库封装了网络套接字函数调用。我们通过修改 *ab* 约 30 行 C 语言代码，就将 *ab* 中使用 APR 库网络 IO 和事件接口的地方替换为 NTSocks 的零拷贝接口（即 *direct-nts*），并通过与基于 NTSocks 的零拷贝 Nignx 协同来实现文件传输大消息场景下 NTSocks 应用性能的评测验证。

5.6.2　性能微基准测试

我们聚焦在两个基本的性能指标：时延和吞吐。对于支持网络套接字抽象的通信系统（即 *NTSocks*、*libvma*、*VPP*、*Linux TCP*），我们实现了统一的基准测试平台来测试性能，该平台通常在充分的“热身”传输次数后，开始采集并聚合汇总运行时的时延、吞吐，POSIX 套接字兼容的通信库可以很容易通过系统变量 *LD_PRELOAD* 加载动态链接库的方式运行基准测试。对于原生 RDMA 性能，我们分别使用 *perftest* 工具提供的 *ib_write_lat* 和 *ib_write_bw*，基于轮询方式来测单边 RDMA Write 原语的网络时延和吞吐。对于裸 NTB 的内存写操作的性能测量，我们模仿 *perftest* 中对单边操作的测评运行流程，实现了基于裸 NTB 轮询传输的基准测试工具 *ntb-bench-tool*。我们将在本节展示主机间通信的性能。

1.　时延

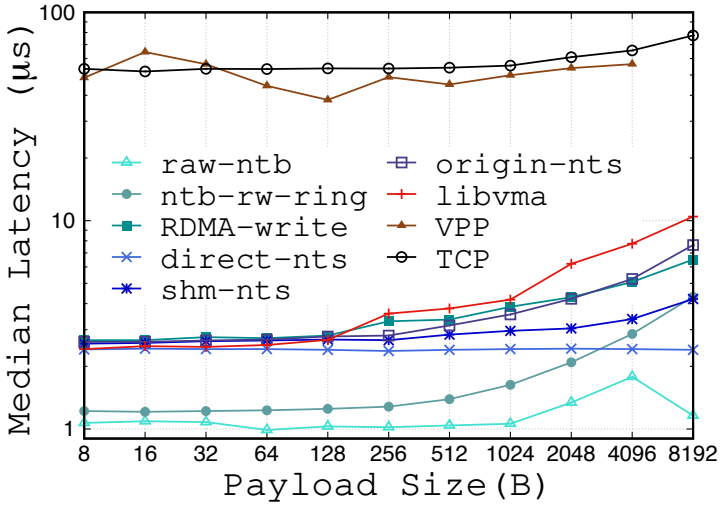


图 5-19　NTSocks 和其他代表性通信栈的中位时延对比

我们构建了类似 *Ping-Pong* 的数据传输程序来测量并记录不同消息大小的往返传输时延，消息大小覆盖了从 8 字节到 8KB 的范围，时延主要包括中位时延（Median Latency）和

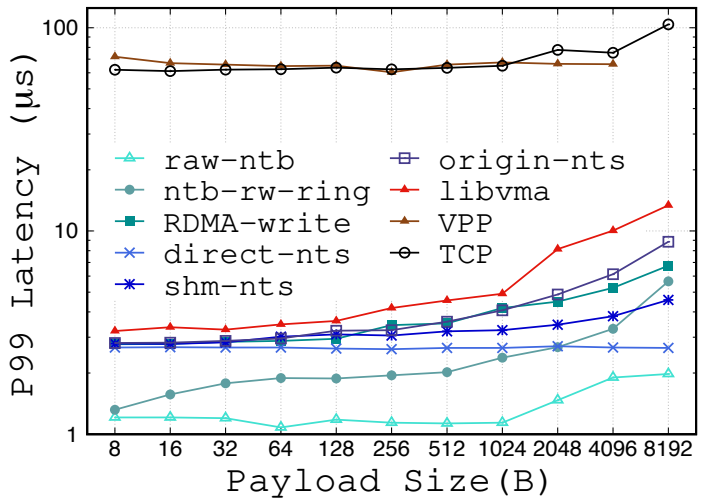


图 5-20　NTSocks 和其他代表性通信栈的 P99 尾时延对比

百分之 99 时延（P99 Latency）。为了保证结果的可靠性，我们针对每个消息大小的时延度量了 100000 次。

图 5-19 和 5-20 分别展示了 *Ping-Pong* 传输的中位时延和 P99 尾时延。我们可以发现裸 NTB 比裸 RDMA（Write 原语传输）有 2.3 到 5.6 倍更低的时延，这主要得益于 PCIe NTB 依赖轻量级的协议栈且避免了 PCIe 协议到网络协议的转译。尽管有套接字到 NTB 原语的转译开销，NTSocks 非侵入式接口 *origin-nts* 却实现了与裸机 RDMA 相似的时延，侵入式接口 *shm-nts* 甚至比裸机 RDMA 实现了高达 35% 的时延降低收益。还可以发现 NTSocks 与裸机 NTB 之间的时延差距较小，在可接受的范围内。然而，即使是非侵入式的 *origin-nts*，套接字到 NTB 原语之间的转译在小消息（<=256 字节）情况下会导致小于 1.7 微秒的额外延迟，在较大消息（>256 字节且 <=4KB）情况下会导致约 2.8 微秒的额外延迟。额外的延迟主要是由于 *libnts* 和 *NTP* 两个组件之间的进程间通信（Inter-Process Communication, IPC）和内存拷贝导致的。在单向传输过程中，NTSocks 需要 2 次 IPCs，不同的模式（即 *origin-nts*、*shm-nts*、*direct-nts*）有不同的内存拷贝次数（如第 5.4.5 节所述）。由于小消息对内存拷贝不敏感而大消息更敏感，三种 NTSocks 模式在小消息下几乎没有时延差距，而在大消息下会存在明显的时延差距，例如，在 4K 字节消息下 *origin-nts* 比 *shm-nts* 有约 1.88 微秒的额外延迟。同样是非侵入式的网络套接字库，*origin-nts* 的时延最低是 RDMA 套接字 *libvma* 的 67%。除此之外，NTSocks 实现了比 Linux 内核套接字和用户态网络栈 *VPP* 一个数量级更低的时延，这是因为众所周知的内核态协议栈的低效率数据包处理[11, 143]，而 *VPP* 是采用面向高吞吐的批处理思想且对时延不友好[74]。需要注意的是，中位时延和 P99 尾时延有类似的表现和成因。

2. 吞吐量

吞吐性能主要体现在针对小消息的往返请求率（即 *Request Rate*）和针对大消息的应用级吞吐（即 *Goodput*）。对于 8 到 256 字节小消息的请求率，我们使用带有 10000 个未决请求（即 Outstanding Requests）的 *Ping-Pong* 数据传输程序，来测量不同网络栈的每秒消息处

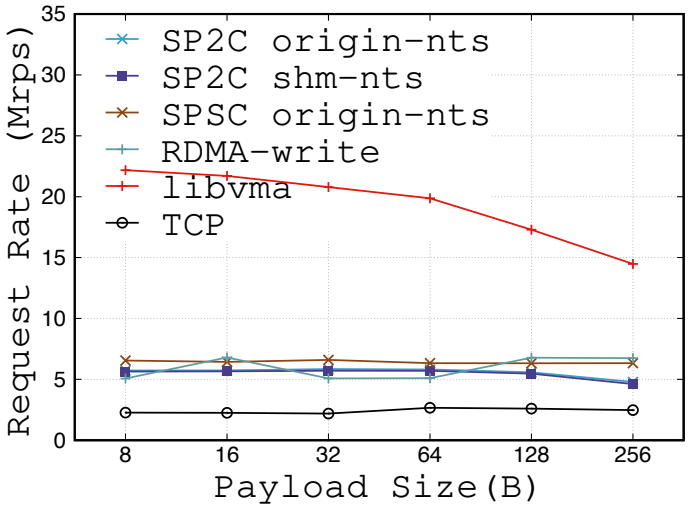


图 5-21　NTSocks 与其他代表性通信栈的小消息请求率对比

理量（即请求率，单位 *Million Requests Per Second*，*Mrps*），并且在客户端采集记录消息请求率。对于 512B 到 4MB 的大消息的吞吐，我们使用客户端到服务端传输数据的单向带宽（单位为 *Gbps*）来测量支持套接字的网络栈，并在服务端记录应用级吞吐。为了保证测量的稳定性和可靠性，每个消息大小的吞吐评测运行时间为 20 秒。

图 5-21 和 5-22 分别展示了小消息的请求率（*Request Rate*）和大消息的吞吐（*Goodput*）。在小消息请求率对比中，如图 5-21 所示，我们发现 NTSocks 在各种模式下均可以实现与裸 RDMA（Write）相近的、比 Linux TCP 高 3 倍左右的、高达 8Mrps 的消息率，并且更平稳，这是因为 NTSocks 依赖的协议栈更轻量级，消除了 PCIe 协议与网络协议之间的转译开销，绕过了系统内核。同时，RDMA 套接字 *libvma* 比 NTSocks 和裸 RDMA 有更高的请求率（2 倍左右），这是因为 *libvma* 采用了小消息批量处理的思想大幅提升吞吐[143, 144]，而随着消息的持续增大，*libvma* 的小消息批量处理优化不再带来吞吐收益，请求率逐渐降低，此时的开销主要由套接字到 RDMA 原语的转译引入的内存拷贝和锁竞争主导[143]，这也是 *libvma* 比 NTSocks 有更高的小消息请求率、更低的应用级吞吐的原因。尽管在一次 *Ping-Pong* 传输中 NTSocks 的非侵入式 *origin-nts* 比侵入式 *shm-nts* 多了两次内存拷贝，但二者请求率非常相近，这是因为内存拷贝对小消息传输引入的开销非常小，可以忽略不计。

在大消息吞吐量对比中，如图 5-22 所示，我们发现 NTSocks 收敛后达到的实测饱和带宽仅为 74.3Gbps，低于裸 RDMA，这是因为我们使用的 NTB 适配器是厂商提供的实验平台，经厂商确认，内置非透明桥芯片颗粒的设计规格是 80Gbps，而发行版商用 NTB 芯片的设计规格均与 PCIe 带宽对齐，我们相信 NTSocks 在商用 NTB 适配器支持下能很容易接近 PCIe 带宽的理论值。NTSocks 比裸 RDMA 有更低的吞吐初始斜率，主要因为：（1）二者不是同一级别的抽象，没有可比性，NTSocks 比裸 RDMA 多了套接字到网络原语的转译开销，这在一次单向传输中引入了额外的 2 次 IPCs 和内存拷贝，而 *libvma* 是基于 RDMA 原语的套接字抽象，同为套接字抽象，NTSocks 单线程吞吐最高是 *libvma* 的 2.9 倍；（2）测评裸 RDMA 的 *ib_write_bw* 和 NTSocks 的吞吐测评程序有所不同，*ib_write_bw* 在发送一个消息后通过轮询完成队列（Completion Queue，CQ）来标识一次传输的结束，且没有应用缓冲区到 RDMA 注册内存区域的拷贝，而原生 NTB 没有硬件级的完成队列机制，需要在软件层通

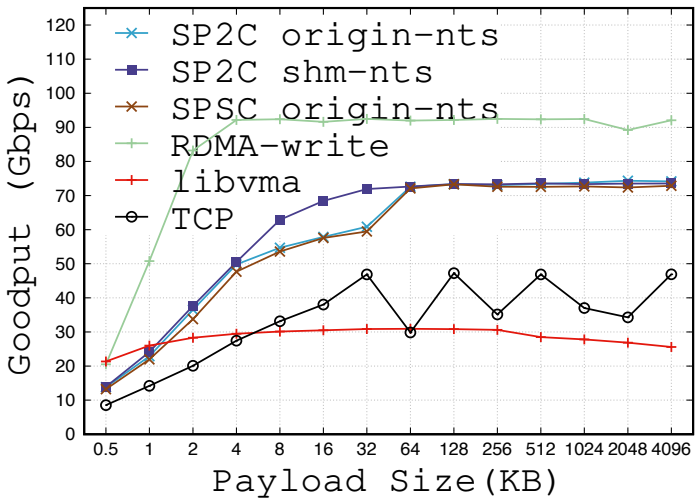


图 5-22　NTSocks 与其他代表性通信栈的大消息应用级吞吐对比

过轮询特定标识位来识别单向传输的结束，且存在应用缓冲区到 NTB 共享内存的拷贝。尽管 NTSocks 不同模式均能收敛到饱和带宽，侵入式 *shm-nts* 比 *origin-nts* 在 0.5 到 32KB 字节的消息范围内有更高的吞吐，且 *shm-nts* 在 32KB 字节消息时先于 *origin-nts* 达到饱和带宽，这主要因为一次单向传输中 *shm-nts* 比 *origin-nts* 少了 2 次应用缓冲区与每连接共享内存之间的内存拷贝。虽然用户通常不希望小消息饱和带宽而更关注时延[173]，当消息大小较小的时候，例如 2KB，NTSocks 仍然可以实现超过饱和带宽一半的吞吐。这种情况下，吞吐主要是由 *NTP* 的单分区限制，可以增加更多的分区来解决吞吐瓶颈。

3. 单向传输性能剖析

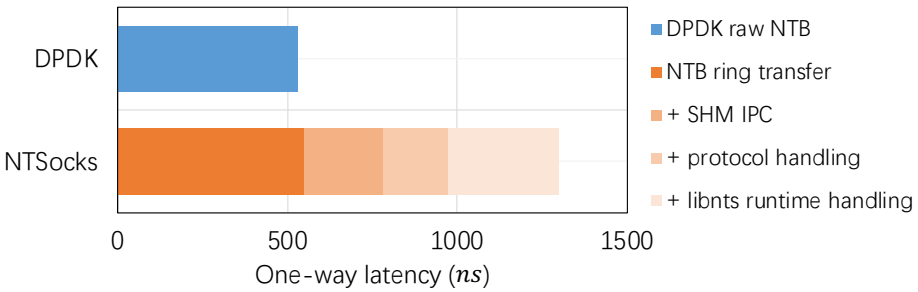


图 5-23　NTSocks（*origin-nts*）8 字节单向传输时延的剖析

我们利用简单的 C 语言 *echo* 基准测试程序量化剖析 NTSocks 网络栈相比于裸 NTB 的关键时延开销，主要包括（1）*NTP* 分区内 NTB 环形缓冲区传输时延、（2）*NTP* 内轻量级协议处理开销、（3）*NTP* 与 *libnts* 应用进程之间基于共享内存的进程间通信开销、（4）*libnts* 运行时处理开销。对于裸 NTB，我们直接利用 DPDK PMD 驱动中的 NTB Write 操作实现了最小化的 echo 程序的客户端和服务端，并记录每个请求的往返时延；对于 NTSocks，我们利用 *origin-nts* 的非侵入式套接字接口实现了 echo 程序的客户端和服务端。NTSocks 中 *NTP* 轻量级协议处理主要指数据包头部元数据，在发送方向上从每连接共享内存队列转发到 NTB 环形缓冲区的封装，在接收方向上从 NTB 环形缓冲区到对应连接共享内存队列的解析；*libnts*

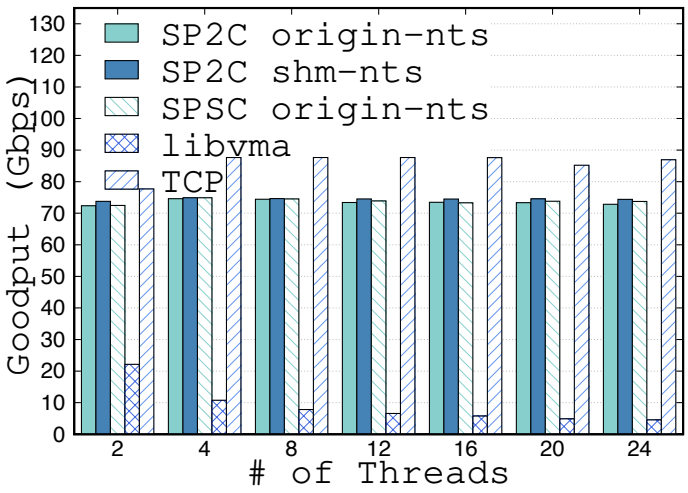


图 5-24　NTSocks 与 libvma、Linux TCP 在并发流量负载下的性能可扩展性对比

运行时处理主要指 *write()* 函数在开始被调用到数据进入连接共享内存队列之前的处理，和 *read()* 函数在开始从连接共享内存队列中取出数据到完整数据被返回之前的处理。我们通过在 NTSocks 组件内部打纳秒级时间戳的方式来记录上述四个部分的时延数据，为了排除打时间戳引入的系统调用开销，我们将每个请求的端到端 *echo* 时延减去时间戳接口系统调用总时间，从而得到最终的每个请求的传输时延。评估中 *Echo* 传输的消息大小为 16 字节，我们通过运行 100000 次取平均值的方式确保结果的准确性。

图 5-23 展示了 NTSocks 一次单向传输的时延开销剖析。我们发现 NTSocks 的分区 NTB 环形缓冲区传输相比裸 DPDK NTB 仅增加了微不足道的时延开销，这主要来源于维护 *write_index* 和 *read_index* 的开销。然而，*NTP* 内部的协议处理开销、*NTP* 和 *libnts* 之间的进程间通信和 *libnts* 运行时处理确实会引入一些时延开销。协议处理开销主要是由于数据包的头部封装和解析，进程间通信开销主要体现在每连接共享内存队列出入队的原子操作和共享内存固有的同步传输开销，*libnts* 运行时处理开销主要由于调用 *write()* 时从连接共享内存池中分配共享内存、将数据从应用缓冲区拷贝到分配的共享内存，以及调用 *read()* 时回收共享内存、将数据从共享内存拷贝到应用缓冲区。虽然上述开销是可以接受的，我们认为这些时延开销在未来可以被进一步降低，比如通过硬件卸载 *NTP* 组件。

5.6.3　扩展性与性能隔离

本节主要验证分析了 NTSocks 相比于其他网络栈有更优良的多核性能缩放能力和多租户性能隔离。

1.　可扩展性

多核性能扩展性是 NTSocks 的一个重要优势，即系统增加额外的 CPU 核心来实现横向扩展。本节中，多核扩展性指随着 CPU 绑定的并发客户端连接的增加，系统总体吞吐量不降低或不受影响。为了验证 NTSocks 的多核扩展性，我们用多个并发绑核线程客户端向同一服务端发送 128KB 的消息请求，并在服务端利用一个背景线程根据每个连接的吞吐实时

统计总体聚合吞吐，线程数量从 2 个变化到 24 个，NTSocks 数据平面设置 4 个分区。由于实验环境中 PCIe NTB 适配器是 80Gbps 的设计规格，而 RoCEv2 ConnectX-5 网卡是 100Gbps，为了统一量化和直观对比，我们对聚合吞吐进行了归一化处理，即实测聚合吞吐与硬件带宽上限的比值，称之为带宽饱和率。

图 5-24 展示了一个非常乐观的结果，即 NTSocks 的聚合带宽饱和率在不同的线程数下均是最高的（在 90% 以上），Linux TCP 其次（在 78% 以上），RDMA 套接字 *libvma* 一直最差（最高仅为 22%）。NTSocks 不同模式的聚合吞吐随着并发线程数的增加均非常稳定，带宽饱和率在 90.4% 至 93.6% 之间，*shm-nts* 比 *origin-nts* 有小于 1.5Gbps 的吞吐差距，这主要得益于侵入式 *shm-nts* 消除了数据路径上应用缓冲区与共享内存队列之间的内存拷贝。RDMA 套接字 *libvma* 的聚合吞吐随着并发线程数的增加呈现先线性降低后趋于稳定的趋势，相比于 2 线程的吞吐，4 线程的吞吐降低到 *1/2*，更多线程数的吞吐降低到 *1/5*，这主要是由于 *libvma* 共享网卡队列上锁的竞争[143, 144]，Linux 套接字的聚合吞吐随着线程数量的增加呈现出先上升后趋于稳定的趋势，带宽饱和率在 77.6% 至 87.6% 之间，尽管 Linux 内核套接字由于 TCP 段卸载（TCP Segmentation Offload，TSO）和发送窗口缩放优化改进了吞吐，但带宽饱和率仍然低于 NTSocks，这主要因为 Linux 内核 TCP 厚重的分层协议栈引入的系统调用开销和多次内存拷贝开销[11]。

2. 性能隔离

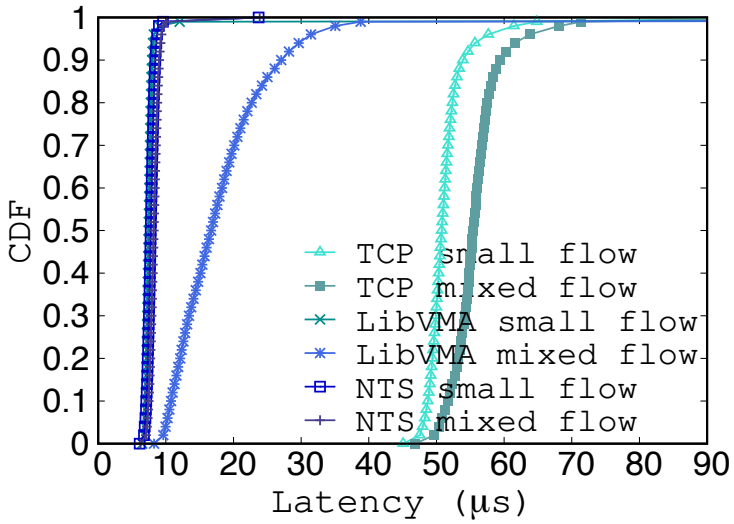


图 5-25　NTSocks 与 Linux TCP 性能隔离效果对比

NTSocks 可以利用分区内、分区间层次化隔离的协同设计来隔离不同流量模式的各种应用或租户之间的性能，确保面向多租户的相对均衡的服务质量。为了验证度量多租户的性能隔离程度，我们利用网络流的时延累积分布和抖动变化来表示流的服务质量和多个流之间的性能干扰程度。我们以一个持续传输 8B 消息的小流（small flow）的时延累积分布作为基准，对比在没有背景流量、有一个 10KB 消息的大流（large flow）作为背景流量两种实验场景下的时延抖动（即时延差距）和尾时延变化，时延抖动或差距越小，背景流量对小流的性能干扰越小，性能隔离效果越好。我们采用了非侵入式接口 *origin-nts* 和两个分区进行

实验。

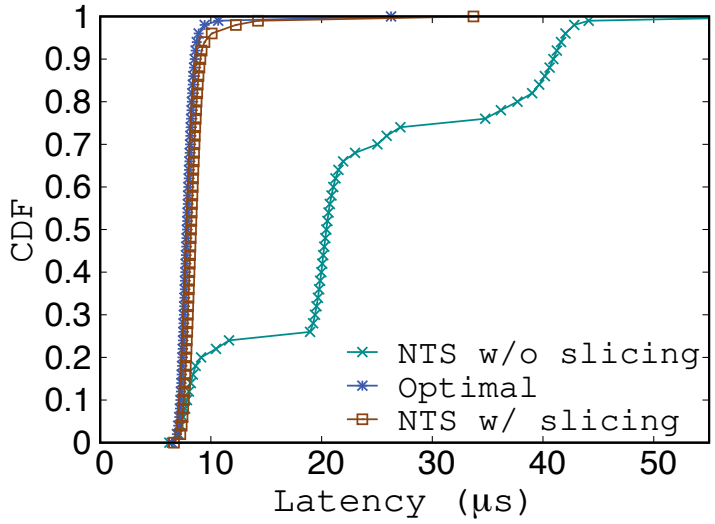


图 5-26　支持消息切片的 NTSocks 可以消除线头阻塞 HOL 问题

图 5-25 展示了 NTSocks 在大小流之间实现了最好的性能隔离效果，比 Linux TCP 套接字有一个数量级更低的时延抖动。NTSocks 小流在有背景流量下实现了与没有背景流量情况下几乎一致的时延分布，二者之间的时延抖动差距分布在 0.36 微秒左右（0.072 至 0.535 微秒之间），P99 尾时延差距仅为 0.53 微秒。图 5-26 展示了在小流、大流复用同一个分区时，没有消息切片的 NTSocks 会导致严重的长尾时延，而支持消息切片的 NTSocks 可以达到接近最优的时延。这得益于 NTSocks 在第 5.4.3 节提出的层次化隔离机制，通过分区内、外的协同设计消除了 NTB 链路复用的线头阻塞，并实现了分区间更好的负载均衡。而 Linux TCP 展示了最糟糕的性能隔离效果，在两种实验场景下的时延抖动差距分布在 4.83 微秒左右（0.759 至 7.216 微秒之间），P99 尾时延差距更是高达 7.216 微秒，存在严重的线头阻塞问题，主要是由于众所周知的内核 TCP 协议栈在数据路径上的低效率（比如频繁的系统调用开销、不同协议层间的协议转译开销、用户态与内核态之间的多次内存拷贝开销等），并进一步加剧了基于滑动窗口限速的慢反应[11, 143]。

5.6.4　PCIe NTB 与 RDMA 混合部署优势

表 5-1　三种场景产生的 PFC 和 CNP 数量比较，PFC 和 CNP 越多，说明网络服务质量越差

	Single	Mixed RDMA	Mixed RDMA/NTB
PFC	20	806	0
CNP	0	4361	0

数据中心网络内混合 PCIe NTB 和 RDMA 的部署方案可以利用 NTSocks 把原来机架内的 RDMA 流量卸载到 PCIe NTB 互连链路上，从而一定程度上缓解高并发机架内、外 RDMA 流量下在 ToR 交换机出端口的 Incast 拥塞问题。为了验证上述优势，我们构建了一个最简化的网络拓扑：服务器 *S1* 和 *S2* 位于同一机架 *R1* 内并通过 ToR 交换机（Mellanox SN2100 型

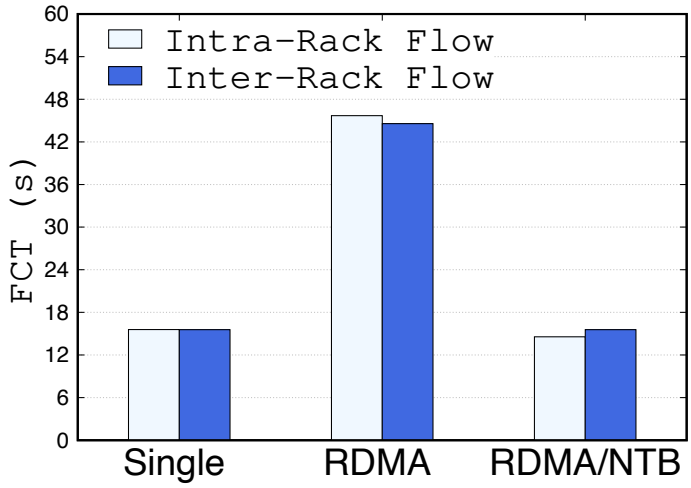


图 5-27　在机架内外独立 RDMA、混合 RDMA、机架内外 NTB/RDMA 混合三种场景的流完成时间对比

号）连接在一个 RDMA 网络，同时 *S1* 和 *S2* 还通过 PCIe NTB 互连，而服务器 *S3* 位于另一个支持 RDMA 的机架 *R2*，两个机架的 ToR 交换机直连在一起，整个 RDMA 网络是 100Gbps 且采用默认的 *DCQCN*[12] 流控，服务器 PCIe 总线均为 GEN 3×16（理论总带宽 128Gbps）。我们把 *S1* 作为目标服务器，*S2* 向 *S1* 打流来仿真机架内的流量（Intra-Rack Flow），*S3* 向 *S1* 打流来仿真机架外的流量（Inter-Rack Flow），流量发送速率均为 56Gbps，且两种流大小一致。在上述拓扑基础上，我们构建了三种场景：（1）分别单独运行机架内、外两种 RDMA 流（Single）；（2）同时运行机架内、外两种 RDMA 流，构成了机架内、外 2:1 RDMA Incast（RDMA）；（3）机架内的 RDMA 流替代为 PCIe NTB 流，同时运行混合的机架内 NTB 流、机架外 RDMA 流（RDMA/NTB）。我们基于 NTSocks 实现了支持限速的 NTB 打流工具，使用 *perftest* 工具中的 *ib_write_bw* 产生 RDMA 流。我们使用 Mellanox *NEO-Host* 工具来监控采集服务端 *S1* 机器上收到的 PFC（Priority-based Flow Control）暂停帧和生成的拥塞通知数据包（Congestion Notification Packet，CNP），使用处理器计数监控工具（Processor Counter Monitor，PCM）来采集 *S1* 上的 PCIe 实时带宽。

如图 5-27 和 5-28 所示，基于 NTSocks 的 NTB 和 RDMA 混合部署，可以在保证机架内、外最低的流完成时间的同时，饱和了目标服务器 *S1* 上的 PCIe 带宽。在表格 5-1 中，基于 NTB 的异构机架内流量卸载（场景 3）同样消除了 RDMA 2:1 Incast（场景 2）拥塞产生的 PFC 和 CNP。具体而言，场景 1 中由于机架内、外 RDMA 流量独立运行且速率小于线速 100Gbps，两种流完成时间都较低，在 15.5 秒左右。场景 2 中由于机架内、外 RDMA 流量同时流向 *S1*，在 *R1* 的 ToR 交换机出端口处形成 2:1 Incast，且总速率 112Gbps 大于 RDMA 线速 100Gbps，两种流竞争出端口带宽，这在 R1 交换机上大量触发了 PFC，并在 *S1* 上生成 CNP 通知源端 *S2*、*S3* 根据 DCQCN 的策略降速，从而导致机架内、外的 RDMA 流完成时间增加了 2 倍（达 45 秒左右），而 RDMA 链路利用率却仅为 60.4%（理想情况下应在 90% 以上）。场景 3 充分体现了 PCIe NTB 和 RDMA 异构混合网络的优势，通过多路径异构网络消除了 RDMA 链路拥塞，机架内流量甚至达到了比场景 2 中更低的流完成时间（1 秒左右），这是因为将机架内 RDMA 流量卸载到 PCIe NTB 链路，避免了两种流量在机顶交换机出端口上的带宽竞争。需要注意的是，NTB 和 RDMA 混合部署在满足如下约束条件（公式 5.1）

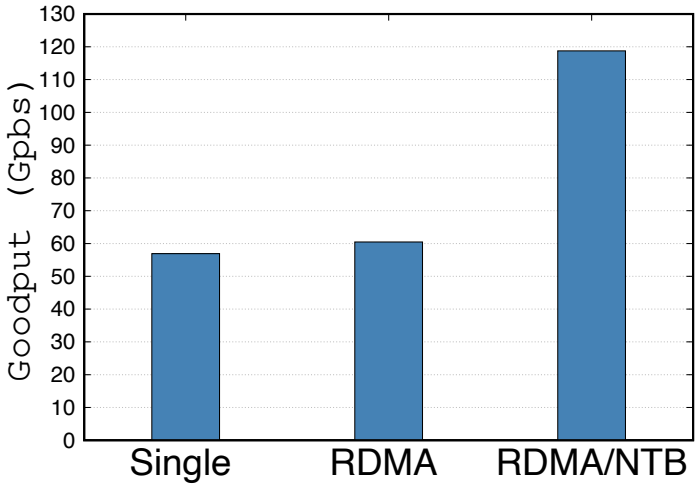


图 5-28　机架内外独立 RDMA、混合 RDMA、机架内外 NTB/RDMA 混合三种场景的目标主机 PCIe 带宽

才能够有效消除混合机架内、外 RDMA 流量的 Incast 拥塞：

$$\begin{aligned} BW_{tor} < BW_{intra} + BW_{inter} < BW_{pcie}, \\ BW_{intra} < BW_{tor}, \\ BW_{inter} < BW_{tor}. \end{aligned} \tag{5.1}$$

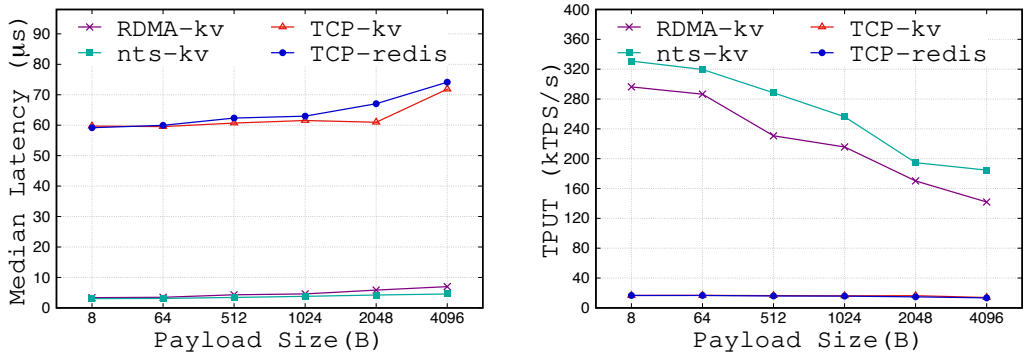
其中，BW_{tor} 指 ToR 交换机线速，BW_{intra} 和 BW_{inter} 分别指机架内、外的 RDMA 流量速率，BW_{pcie} 指目标服务器的 PCIe 带宽上限。

5.6.5　实际应用性能

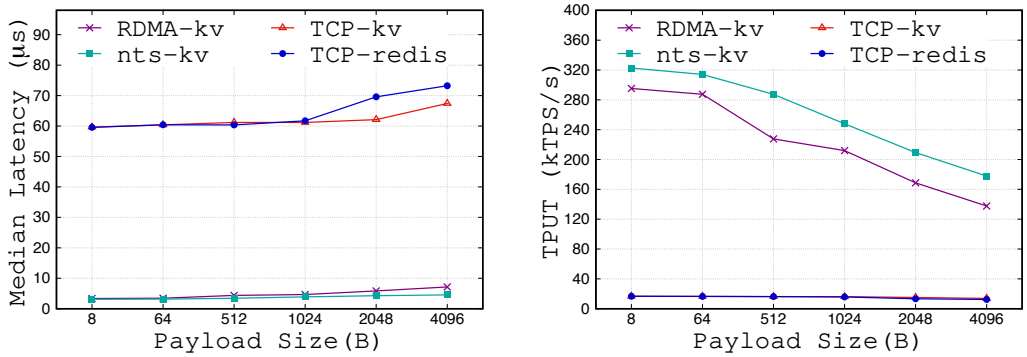
本节展示了 NTSocks 可以在不需要改动应用代码或最小化的代码修改情况下就可以明显地提升实际系统的端到端性能。我们使用键值存储系统和 Nginx 文件传输服务来分别展示了 NTSocks 在小消息、大消息实际应用中的性能收益。需要注意的是，在应用系统上很小的性能收益就意味着通信性能的很大提升[151]。

1.　键值存储系统

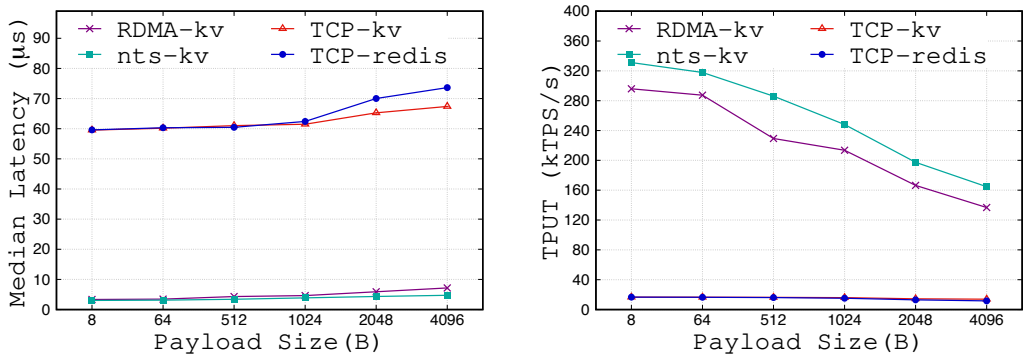
本节主要关注在 NTSocks 在时延敏感的小消息场景下的性能收益以及对不同工作负载的泛化能力（Workload-Independent）。我们使用不同类型的 YCSB 工作负载来评测内存键值存储系统的实际性能，将单线程模型驱动的 KVS 系统运行在两台服务器上，分别运行 KVS 服务端和 YCSB 客户端。在实验中，内存键值存储系统主要包括基于非侵入式 NTSocks 的 KVS（*nts-kv*）、基于 RDMA 套接字 libvma 库的 KVS（*RDMA-kv*）、基于 Linux TCP 的 KVS（*TCP-kv*）和 Redis（*TCP-redis*）。YCSB 工作负载混合了不同比例的各种请求类型，包括：（1）100% 的 *Read* 请求（R100）；（2）95% 的 *Read* 请求和 5% 的 *Insert* 请求（R95I5）；（3）90% 的 *Read* 请求和 10% 的 *Update* 请求（R90U10）；（4）50% 的 *Read* 请求和 50% 的 *Update* 请求（R50U50）；（5）50% 的 *Read* 请求和 50% 的 *Read-Modify-Write* 请求（R50M50）。不同



(a) YCSB-R100



(b) YCSB-R95I5

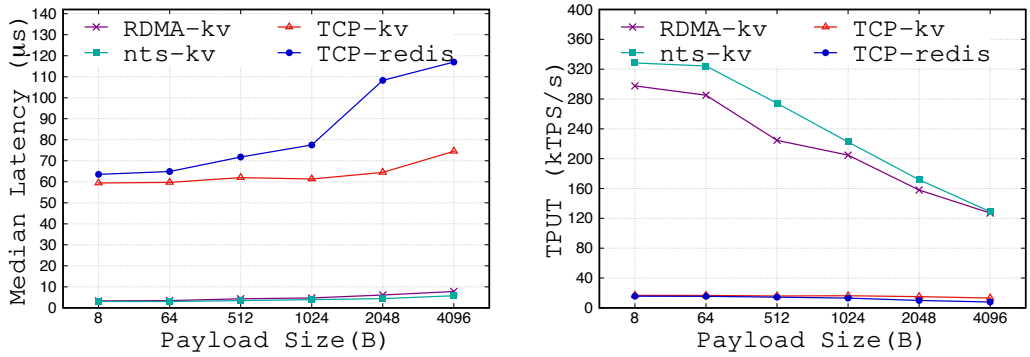


(c) YCSB-R90U10

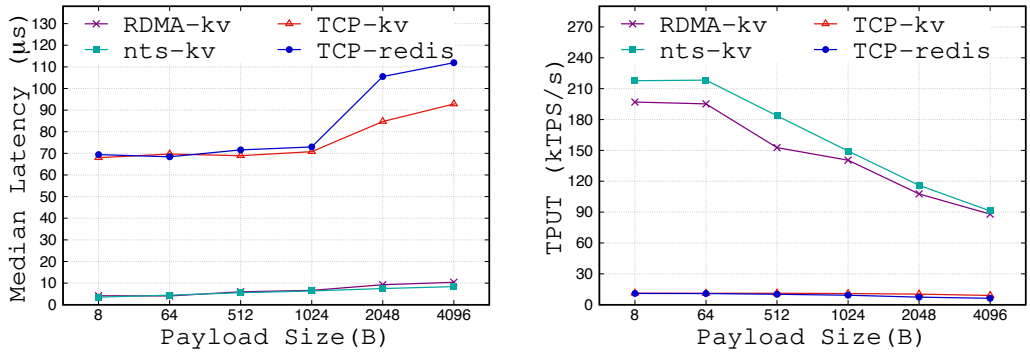
图 5-29　不同 YCSB 工作负载下键值存储系统的中位时延和吞吐比较（1）

的负载类型产生了不同的流量模式，数据负载采用基于 *zipfian* 请求分布，每个数据记录包含 10 个字段（即值）和对应键，控制字段大小分别为 8B、64B、512B、1KB、2KB 和 4KB，从每个负载类型均运行共 10 万次的数据操作。主要性能指标体现在每种 YCSB 工作负载类型下 KVS 的中位时延和吞吐。

图 5-29 和 5-30 展示了不同 YCSB 工作负载下的性能结果。我们可以总结为两点。第一，传输层的性能确实是内存键值存储中的一个潜在系统瓶颈。对于键值操作性能，NTSocks（即 *nts-kv*）和 libvma （即 *RDMA-kv*）均比 Linux TCP 有一个数量级更好的端到端时延和吞吐。同为套接字抽象，即使 libvma 已经利用 RDMA 的性能优势大幅加速了键值存储的性能，NTSocks 仍然实现了比 libvma 了高达 57.6% 更低的中位时延，高达 30.1% 更高的键值



(a) YCSB-R50U50



(b) YCSB-R50M50

图 5-30　不同 YCSB 工作负载下键值存储系统的中位时延和吞吐比较（2）

IOPS。第二，在不同的工作负载和键值大小下，NTSocks 性能确实优于 RDMA 套接字 libvma 库，对不同流量模式具有泛化能力。例如，图 5-29(a) 展示了在 YCSB R100 负载下 NTSocks 比 libvma 有 10.4%~52.7% 更低的中位时延，11.5%~30.1% 更高的吞吐。图 5-29(b) 展示了在 YCSB R95I5 负载下 NTSocks 比 libvma 有 8.7%~57.6% 更低的中位时延，9.2%~29.2% 更高的吞吐。图 5-29(c) 展示了在 YCSB R90U10 负载下 NTSocks 比 libvma 有 10.3%~51.7% 更低的中位时延，10.5%~24.7% 更高的吞吐。图 5-30(a) 展示了在 YCSB R50U50 负载下 NTSocks 比 libvma 有 10.2%~38.4% 更低的中位时延，1.4%~22.1% 更高的吞吐。图 5-30(b) 展示了在 YCSB R50M50 负载下 NTSocks 比 libvma 有 4.3%~24.2% 更低的中位时延，3.5%~20.2% 更高的吞吐。这主要得益于 NTSocks 实现了数据路径上套接字到 PCIe NTB 原语的最小化转译开销，同时也说明了 NTSocks 工作负载无关的泛化能力。

2. 基于 Nginx 的 HTTP 文件传输服务

我们在两台服务器上构建一个典型的内存文件传输 Web 服务系统并进行性能测量，用于验证 NTSocks 在大消息传输场景下的性能收益。我们在一台机器上运行 Nginx 实例作为内存文件服务器，在另一台机器上运行 Apache 基准测试工具 *ab* 用于发送不同大小的文件请求，二者之间采用保活（Keep-Alive）HTTP 连接通信，文件大小范围是从 8K 到 32MB，在 *ab* 中测量从发送文件请求到收到响应的文件传输时延。libvma 由于 *fork* 无法直接运行在 Nginx 上。

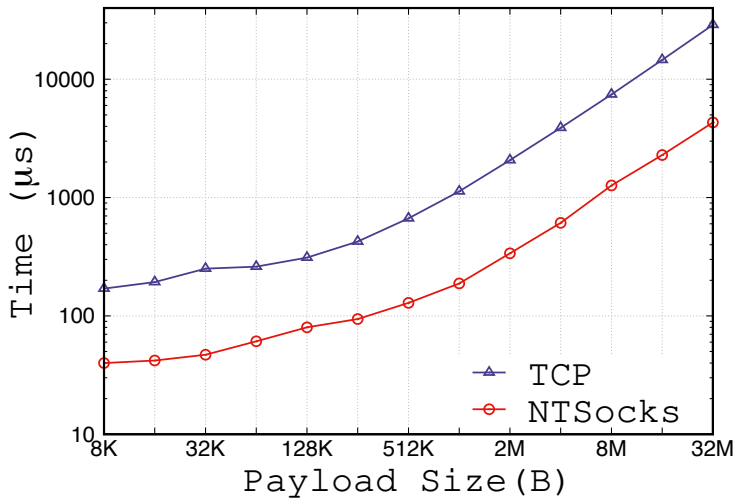


图 5-31　Nginx HTTP Web 服务器的端到端文件请求处理性能

图 5-31 展示了 Linux TCP 的文件传输时延是 NTSocks 的 3.88~6.69 倍，这主要得益于 NTSocks 的零拷贝设计。随着文件大小的增加，内存拷贝开销越大，NTSocks 与 Linux TCP 之间文件传输时延的差距也逐渐增大。总而言之，NTSocks 在大消息传输的应用场景也能有显著的性能收益。

5.7　讨论

RDMA 和 PCIe NTB 协同设计在一个统一的框架中。我们通过实验验证了 NTSocks 的流与 RDMA 的流混合部署在总流量小于 PCIe 带宽限制的情况下几乎互不干扰。在高并发的机架内、机架外流量负载情况下，现有的 RDMA 部署会遇到 ToR RDMA 交换机出端口 Incast 拥塞，进而导致普遍的 PFC 风暴和拥塞广播问题，极高的（尾）时延严重降低了传输服务质量。而将现有机架内 RDMA 流量卸载到机架内的 PCIe NTB 链路上，可以有效地缓解上述 RDMA 出端口 Incast 拥塞问题，并使得目标主机上的 PCIe 带宽达到饱和。但这种粗粒度的 RDMA 和 PCIe NTB 混合部署无法实现基于异构多路径的细粒度网络流调度，从而在总流量大于 PCIe 带宽的情况下会互相干扰，既降低了性能又无法使得链路传输资源充分饱和，这需要 RDMA 和 PCIe NTB 传输在同一架构内的协同设计才能实现互相的负载感知、细粒度的流量控制，以使 PCIe 总带宽饱和的同时，提高异构多路径资源的复用率。我们把该协同设计作为未来工作的一部分。

NTsocks 加速边缘云网络。边缘云弥合了边缘终端设备到云端数据中心的长距离传输的不足，以更靠近边缘设备的方式加速边缘海量数据的计算任务（即边缘计算），提供更实时的分析或在线服务。因此边缘云内服务器之间的高质量协同计算需要高速网络提供极低的时延。而边缘云通常是几台至十几台服务器互连的一个机架，服务器之间的横向网络流量主要以机架内通信传输为主，且组网简单，这是 PCIe NTB 互连的一个天然适配场景，由单个 PCIe NTB 集群交换机连接的服务器集群组成了一个超低时延、高吞吐的边缘云网络基础设施。这是我们未来探索的一个重要方向。

NTSocks 的扩展友好性。许多现代机架使用多个片上系统 SoCs 作为高性能网络或存储

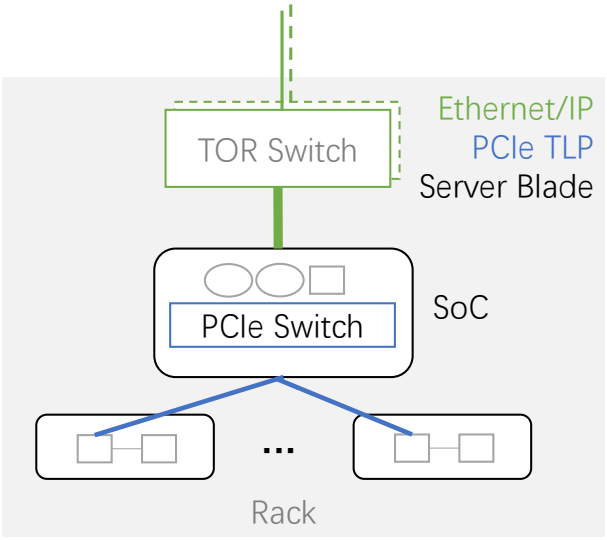


图 5-32　基于 SoCs 的、PCIe 互连与以太网混合部署的机架内网络是未来的一个潜在机架级网络形态

IO 加速器，且对时延有近乎苛刻的要求，例如 UCloud 中基于 Mellanox DPU（Data Processor Unit）的裸金属云服务或阿里巴巴盘古云存储前端的神龙加速卡[137]。因而一个机架内对 PCIe 连接的外围设备（如 SoCs）跨机器间的高速访问是常见且重要的，也是 PCIe NTB 可以实现比 RDMA 更容易控制的典型场景。该场景下，对 PCIe 连接的外围 PCIe 设备基于 RDMA 单向访问的关键数据路径是本地 CPU–> 本地 PCIe–> 本地 RDMA 网卡–>Ethernet 网络–> 远端 RDMA 网卡–> 远端 PCIe–> 远端 CPU–> 远端 PCIe–> 远端目标 PCIe 设备，而基于 PCIe NTB 单向访问的关键路径是本地 CPU–>PCIe 互连网络–> 远端目标 PCIe 设备。可以明显地发现，对 PCIe 连接外围设备的基于 PCIe NTB 的访问可以消除对以太网链路和远端主机 CPU 的依赖，避免了复杂的数据路径，从而实现了比 RDMA 更极致的低时延。我们认为未来机架内网络架构的潜在网络形式是基于片上系统的、以 PCIe 互连为主导、以以太网为对外通路的架构，如图 5-32 所示，片上系统作为机架对外网关，直接与机顶以太网交换机相连，而机架内的其他机器利用一个 PCIe 交换机与该片上系统连通，PCIe NTB 互连对于这种机架网络架构有天然的扩展友好性，并提供超低的传输时延，我们将在未来的工作中展开研究。

5.8　本章小结

在本节中，我们介绍了 NTSocks 系统，这是一种基于 PCIe 非透明桥互连的用户态机架级的通信架构，可提供机架级系统所需的满足兼容易用性、高性能、多核扩展性、多租户性能隔离和高效资源共享的网络功能。NTSocks 系统展示了灵活的用户级间接层可以实现接近裸 NTB 的性能，同时解决了原生用户态 NTB 的三个关键问题：抽象不匹配、缺乏多核扩展的数据平面、多应用之间的资源共享和性能隔离。对实际应用程序和微基准测试的评估结果表明，NTSocks 达到的性能可与原生 NTB 相媲美，并且远优于现有的 Linux 和 RDMA 套接字实现。我们将开源 NTSocks 的原型系统实现。

第 6 章　总结与展望

6.1　研究工作总结

高性能内核旁路网络的快速发展正加速着分布式系统的变革。而随着数据规模的指数级递增，数据中心规模的分布式系统更加依赖于越来越频繁的多节点数据通信，这又促使了内核旁路网络的广泛部署与应用。内核旁路网络不仅突破了传统操作系统内核 TCP/IP 网络栈的带宽和时延约束，而且在资源开销、数据零拷贝等方面也具备明显优势。然而，一方面，大规模数据处理和新型计算存储硬件的广泛使用，使得传统分布式系统的性能瓶颈转移到网络。另一方面，内核旁路网络依赖全新的通信原语抽象，与传统的通用网络抽象不匹配，这阻碍了分布式系统对内核旁路网络性能优势的发挥。同时，传统内核态 TCP/IP 协议栈已主导网络通信数十年，分布式系统过多地针对内核态网络栈进行优化设计，这已经不再适用于内核旁路网络。随着低时延、高带宽的内核旁路网络在数据中心的规模化部署，如何充分利用内核旁路网络的特性和通信原语抽象增强分布式系统的性能，以发挥出内核旁路网络的巨大性能潜力，是一个亟需应对的重要问题。

针对该问题，本文系统性地探索了利用内核旁路网络构建以高性能、资源高效、兼容扩展为优化目标的分布式系统。围绕上述目标，本文创新性地提出了两种内核旁路网络加速系统性能的设计思想与方法，即：（1）基于自上而下的系统适配通信原语，重新设计分布式系统通信层以适用于内核旁路网络；（2）基于自下而上的通信原语适配系统，设计基于内核旁路网络的统一轻量级通信协议栈以适用于分布式系统。本文进一步从网络与系统协同设计、软硬件协同设计两个角度，对内核旁路网络加速吞吐/内存开销敏感的系统、时延/CPU 开销敏感的系统、高性能用户态协议栈三个方面关键技术展开了研究：

1. 在优化分布式系统的吞吐和内存开销方面，本文提出了 RDMA 增强的高性能、内存高效、高可用的文档数据模型及其对应的 NoSQL 系统实现 RMongo。RMongo 在利用 RDMA 网络的单边通信原语特性为 NoSQL 文档数据模型提供高性能增删改查操作的同时，力求实现数据模型的低内存开销以及文档 NoSQL 系统的高可用。RMongo 的设计体现在性能、低内存开销与可用性三个方面。在性能方面，RMongo 利用带有即时数据的 RDMA 写入原语特性扩展了面向文档数据模型的增删改查操作语义，利用 RDMA 写入原语内置的即时数据在 NoSQL 客户端和服务端之间快速同步数据操作的请求控制参数和文档数据模型的元数据信息，提升了端到端数据操作性能，并进一步利用单边 RDMA 原语绕过远端 CPU 的特性，将服务端的数据处理负载转移摊销到客户端，从而改进了文档 NoSQL 服务端在高并发请求下性能的扩展性。在低内存开销方面，RMongo 提出了负载感知的 RDMA 缓冲区注册机制，在满足按需分配内存的同时避免了频繁注册/注销 RDMA 内存区域的 CPU 开销。在可用性方面，RMongo 首先针对传输链路的容错性，提出了 RDMA 上下文检测算法，决策两个存储节点之

间使用 RDMA 或 TCP/IP 通信；其次，RMongo 针对文档数据模型的高可用，结合 RDMA 异步完成事件通道，提出了基于单边 RDMA 写原语的多切片节点之间操作日志同步协议，通过单边 RDMA 原语重新设计初始同步和稳态复制两个阶段。实验表明，RMongo 比传统基于 Linux TCP 的文档 NoSQL 系统提升了高达 30% 的端到端数据操作吞吐量。

2. 在优化分布式系统的时延和 CPU 开销方面，本文提出了 RDMA 驱动的高性能、CPU 高效、可扩展的分布式共识协议及其对应的区块链系统实现 BoR。传统分布式共识协议既需要频繁的多机器之间点到点通信来完成一致性共识确认，又需要消耗大量的计算资源进行共识哈希计算，而且传统分布式系统忽略了内核态 TCP 网络栈数据路径上引入的较大 CPU 开销对分布式共识计算的影响，这带来了不可忽视的通信时延开销和明显的 CPU 资源开销问题。针对该问题，BoR 提出了基于 RDMA 的共识同步管理机制，结合单边 RDMA 写入原语的超低时延和 CPU 旁路特性，利用基于 RDMA 完成事件通道的异步 I/O 事件模型重新设计了驱动分布式共识协议的点到点通信，既保证了分布式区块链系统内更低的区块和交易共识同步时延，又最小化了频繁通信引入的 CPU 开销。同时，BoR 利用 RDMA 共享接收队列和混合通信原语设计了新节点引导协议，通过多连接复用 RDMA 共享接收队列、双边原语交换 RDMA 内存元数据、单边原语收发控制和数据消息，有效规避 RDMA 硬件缓存资源的局限性，降低了新节点初始化同步历史区块数据的时延，提高了共识网络的扩展性。实验表明，在不同规模的工作负载下，相比于基于 Linux TCP 的区块链系统，BoR 的新节点共识过程降低了 CPU 开销达 26.4%，降低了区块同步时延达 20.2%。

3. 在分布式系统依赖的网络协议栈方面，本文提出了基于 PCIe 互连的高性能用户态通信架构及其系统实现 NTSocks。与基于内核态网络栈优化的系统通信不同，基于内核旁路网络的通信协议栈设计不仅需要考虑内核旁路网络资源的局限性和多租户性能隔离挑战，还需要考虑内核旁路网络的异构性。为此，基于数据和控制平面分离的思想，NTSocks 提出了一个高性能、软硬件协同的用户级间接层，既屏蔽了底层高速网络 PCIe 互连和 RDMA 异构性，又实现了兼容、多核扩展、性能隔离的网络功能，有效隔离不同流量模式的系统性能。考虑到 PCIe NTB 互连比 RDMA 少了 PCIe 协议与网络协议之间的转译开销，且组网简单，本文设计了机架内和机架间的通信分别基于 PCIe NTB 和 RDMA 实现的混合部署方案，实现了接近纳秒级的机架内通信时延，有效缓解了机架内 RDMA 拥塞问题。通过设计兼容的类套接字抽象，NTSocks 以用户态运行时库的形式解决了原生 PCIe NTB 抽象不匹配的问题。为了数据平面的多核扩展性，NTSocks 基于并行化思想实现了 CPU 核心驱动的数据平面模型。为了公平高效的资源共享，NTSocks 设计实现了面向多租户的性能隔离机制。实验表明，相比 Linux TCP 和 RDMA 套接字，NTSocks 降低微基准测试时延分别到 1/32 和 1/4.4，降低键值存储系统的端到端时延分别到 1/24.5 和 1/1.58，降低了 Nginx HTTP 文件服务响应时延到 1/6.7。

6.2　未来研究工作展望

本文从网络与系统协同和软硬件协同的角度，对基于内核旁路网络的吞吐/内存敏感型系统增强、时延/CPU 敏感型系统增强和高性能协议栈设计三个方面关键技术展开研究，以探究适用于内核旁路网络 RDMA 和 PCIe NTB 的分布式系统增强和轻量级协议栈设计。在未来工作中，我们将继续深入探究内核旁路网络在系统增强和协议栈设计的其它潜在问题，并尝试应用于实际生产环境中。

以本文研究成果作为基础，以下几个方面的研究方向和内容有很大的潜在研究价值，可以在未来工作中进一步展开研究：

1. **异构内核旁路网络协同的流量控制机制：**多种异构内核旁路网络在数据中心网络的混合部署已经存在，用于加速不同应用场景的系统服务，已实现最佳的服务质量。然而，多种异构内核旁路网络之间缺少统一架构下的协同流量控制，导致一些物理上共享或交叉的链路存在瓶颈，例如 RDMA 与 PCIe NTB 的混合部署，在双方高并发流量下会在主机端上的 PCIe 总线上形成瓶颈。因而，在基于异构混合内核旁路网络的流量控制是一个值得研究的重要问题。

2. **用户态与内核态协同的协议栈设计：**通信协议栈向下管理网络设备和不同类型的内核旁路网络，向上为分布式系统提供网络功能抽象。本文从内核旁路网络的发展推动协议栈变革的角度，研究了基于内核旁路网络的高性能用户态协议栈设计。另一方面，传统内核态协议栈在安全可靠方面的天然优势，也将弥补用户态协议栈在安全隔离方面的不足。例如，将网络协议栈中的控制面功能设计实现到内核态以保证控制路径的安全隔离性，将网络协议栈的数据面功能设计实现到硬件和用户态以保证数据路径的高性能和扩展性。因此，通信协议栈的设计可以采用用户态与内核态协同的方式推动进一步研究。

3. **基于在网计算（In-network computing）的分布式系统设计：**许多可编程智能网卡、片上系统（SoC）、可编程交换机等可编程硬件已经逐渐应用于数据中心，提供了在网计算能力，为分布式系统在负载均衡方面的优化带来新的机遇。如何将分布式系统中基于软件实现的负载均衡器卸载到具备在网计算能力的可编程硬件，分别在数据平面设备的可编程交换机、端上的可编程网卡上实现面向机架内多主机、主机内多核的负载均衡，从而消除软件负载均衡的潜在 CPU 瓶颈，是一个值得深度探索和研究的问题。

4. **基于可编程拥塞控制的有损 RDMA 协议栈：**商用 RDMA 在数据中心已逐渐规模化部署，且默认依赖于基于优先级流量控制（Priority-based Flow Control，PFC）的无损网络，这在达到一定规模情况下很容易出现 PFC 风暴和大幅降速，限制了商用 RDMA 的部署规模和服务质量，也使得有损 RDMA 协议栈的研究至关重要。另一方面，新一代 RDMA 网卡上可编程拥塞控制（Programmable Congestion Control，PCC）的支持，为设计适应于各种工作负载的高性能有损 RDMA 协议栈带来了巨大机遇。例如，默认固化到 RDMA 硬件的 DCQCN 协议，在突发流量情况下普遍地面临收敛慢的问题，基于软件实现的 RDMA 拥塞控制引入了较大的 CPU 开销，而 PCC 可以允许灵

活地设计适应于不同工作负载的数据面拥塞控制机制，从而在满足收敛性要求的同时保证低开销。因此，高性能有损 RDMA 协议栈可结合新一代 RDMA 网卡的可编程拥塞控制特性展开进一步研究。

参考文献

[1] CAO W, LIU Z, WANG P, et al. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database[J]. Proceedings of the VLDB Endowment, 2018, 11(12): 1849-1862.

[2] CHEN Z, HE Q, MAO Z, et al. A study on the characteristics of douyin short videos and implications for edge caching[C]//Proceedings of the ACM Turing Celebration Conference-China. New York: ACM, 2019: 1-6.

[3] CLOUD G. Separation of storage and compute in bigquery[EB/OL]. 2021. https://azure.microsoft.com/en-us/services/blockchain-service/.

[4] GAO Y, LI Q, TANG L, et al. When cloud storage meets {RDMA}[C]//18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21). Berkeley: USENIX, 2021: 519-533.

[5] ZHU H, KAFFES K, CHEN Z, et al. Racksched: A microsecond-scale scheduler for rack-scale computers[C]//14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20). Berkeley: USENIX, 2020: 1225-1240.

[6] DEAN J, BARROSO L A. The tail at scale[J]. Communications of the ACM, 2013, 56(2): 74-80.

[7] GRAEFE G, VOLOS H, KIMURA H, et al. In-memory performance for big data[J]. 2014.

[8] AZURE M. Azure blockchain service[EB/OL]. 2021. https://cloud.google.com/blog/products/bigquery/separation-of-storage-and-compute-in-bigquery.

[9] MONGODB I. Mongodb[EB/OL]. 2009. https://www.mongodb.com/.

[10] GUO C, WU H, DENG Z, et al. Rdma over commodity ethernet at scale[C]//Proceedings of the 2016 ACM SIGCOMM Conference. New York: ACM, 2016: 202-215.

[11] JEONG E, WOOD S, JAMSHED M, et al. mtcp: a highly scalable user-level {TCP} stack for multicore systems[C]//11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14). Berkeley: USENIX, 2014: 489-502.

[12] ZHU Y, ERAN H, FIRESTONE D, et al. Congestion control for large-scale rdma deployments[J]. ACM SIGCOMM Computer Communication Review, 2015, 45(4): 523-536.

[13] PROJECT D. Ntb rawdev driver[EB/OL]. 2020. https://doc.dpdk.org/guides/rawdevs/ntb.html.

[14] ICS D. Pci express gen 3 x16 ntb host adapter[EB/OL]. 2019. https://www.dolphinics.com/products/MXH830.html.

[15] CHEN Y, LU Y, SHU J. Scalable rdma rpc on reliable connection with efficient resource sharing[C]//Proceedings of the Fourteenth EuroSys Conference 2019. New York: ACM, 2019: 1-14.

[16] WIKIPEDIA. Large send offload[EB/OL]. 2020. https://en.wikipedia.org/wiki/Large_send_offload.

[17] MELLANOX TECHNOLOGIES I. Rdma aware programming user manual[EB/OL]. 2015. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[18] KALIA A, KAMINSKY M, ANDERSEN D. Datacenter rpcs can be general and fast[C]//16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19). Berkeley: USENIX, 2019: 1-16.

[19] SULLIVAN M J. Intel xeon processor c5500/c3500 series non-transparent bridge[J]. Technology@ Intel Magazine, 2010.

[20] YU X. Ntb: Add support for amd pci-express non-transparent bridge[EB/OL]. 2016. https://lwn.net/Articles/672752/.

[21] TECHNOLOGIES P. Multi-host system and intelligent i/o design with pci express[EB/OL]. 2005. https://lwn.net/Articles/672752/.

[22] BROADCOM. Pex8733, pci express gen 3 switch, 32 lanes, 18 ports[EB/OL]. 2011. https://docs.broadcom.com/docs/12351852.

[23] MARKUSSEN J, KRISTIANSEN L B, STENSLAND H K, et al. Flexible device sharing in pcie clusters using device lending[C]//Proceedings of the 47th International Conference on Parallel Processing Companion. New York: ACM, 2018: 1-10.

[24] MARKUSSEN J, KRISTIANSEN L B, HALVORSEN P, et al. Smartio: Zero-overhead device sharing through pcie networking[J]. ACM Transactions on Computer Systems (TOCS), 2021, 38(1-2): 1-78.

[25] PROJECT D. Ntb sample application[EB/OL]. 2020. https://doc.dpdk.org/guides/sample_app_ug/ntb.html.

[26] LI Y, MIAO R, LIU H H, et al. Hpcc: high precision congestion control[M]//Proceedings of the ACM Special Interest Group on Data Communication. New York: ACM, 2019: 44-58.

[27] PFISTER G F. An introduction to the infiniband architecture[J]. High Performance Mass Storage and Parallel I/O, 2001, 42: 617-632.

[28] RASHTI M J, AFSAHI A. 10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g[C]//2007 IEEE International Parallel and Distributed Processing Symposium. Piscataway: IEEE, 2007: 1-8.

[29] BECK M, KAGAN M. Performance evaluation of the rdma over ethernet (roce) standard in enterprise data centers infrastructure[C]//Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching. [S.l.]: International Teletraffic Congress, 2011: 9-15.

[30] WIKIMEDIA FOUNDATION I. Rdma over converged ethernet[EB/OL]. 2018. https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet.

[31] MITTAL R, SHPINER A, PANDA A, et al. Revisiting network support for rdma[C]// Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. New York: ACM, 2018: 313-326.

[32] MACARTHUR P. Userspace rdma verbs on commodity hardware using dpdk[C]//2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI). Piscataway: IEEE, 2017: 103-110.

[33] GRANT R E, RASHTI M J, BALAJI P, et al. Scalable connectionless rdma over unreliable datagrams[J]. Parallel Computing, 2015, 48: 15-39.

[34] LU Y, CHEN G, LI B, et al. Multi-path transport for {RDMA} in datacenters[C]//15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18). Berkeley: USENIX, 2018: 357-371.

[35] ISLAM N S, SHANKAR D, LU X, et al. Accelerating i/o performance of big data analytics on hpc clusters through rdma-based key-value store[C]//2015 44th International Conference on Parallel Processing. Piscataway: IEEE, 2015: 280-289.

[36] WANG Y, MENG X, ZHANG L, et al. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores[C]//Proceedings of the ACM Symposium on Cloud Computing. New York: ACM, 2014: 1-13.

[37] SHANKAR D, LU X, ISLAM N, et al. High-performance hybrid key-value store on modern clusters with rdma interconnects and ssds: non-blocking extensions, designs, and benefits[C]//2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Piscataway: IEEE, 2016: 393-402.

[38] KALIA A, KAMINSKY M, ANDERSEN D G. Using rdma efficiently for key-value services[J]. ACM SIGCOMM Computer Communication Review, 2015, 44(4): 295-306.

[39] WANG Y, ZHANG L, TAN J, et al. Hydradb: A resilient rdma-driven key-value middleware for in-memory cluster computing[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Piscataway: IEEE, 2015: 1-11.

[40] CASSELL B, SZEPESI T, WONG B, et al. Nessie: A decoupled, client-driven key-value store using rdma[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(12): 3537-3552.

[41] JOSE J, SUBRAMONI H, LUO M, et al. Memcached design on high performance rdma capable interconnects[C]//2011 International Conference on Parallel Processing. Piscataway: IEEE, 2011: 743-752.

[42] MITCHELL C, GENG Y, LI J. Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store[C]//Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13). Berkeley: USENIX, 2013: 103-114.

[43] YANG M, YU S, YU R, et al. Innercache: A tactful cache mechanism for rdma-based key-value store[C]//Web Services (ICWS), 2016 IEEE International Conference on. Piscataway: IEEE, 2016: 646-649.

[44] DRAGOJEVIĆ A, NARAYANAN D, CASTRO M, et al. Farm: Fast remote memory[C]// 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14). Berkeley: USENIX, 2014: 401-414.

[45] SHI J, YAO Y, CHEN R, et al. Fast and concurrent {RDF} queries with rdma-based distributed graph exploration[C]//12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). Berkeley: USENIX, 2016: 317-332.

[46] KIM J, KANG S, AHN H, et al. Architecture reconstruction and evaluation of blockchain open source platform[C]//Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. New York: ACM, 2018: 185-186.

[47] ZHENG P, ZHENG Z, LUO X, et al. A detailed and real-time performance monitoring framework for blockchain systems[C]//Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. New York: ACM, 2018: 134-143.

[48] PASS R, SEEMAN L, SHELAT A. Analysis of the blockchain protocol in asynchronous networks[C]//Annual International Conference on the Theory and Applications of Cryptographic Techniques. Berlin: Springer, 2017: 643-673.

[49] PAPADIS N, BORST S, WALID A, et al. Stochastic models and wide-area network measurements for blockchain design and analysis[C]//IEEE INFOCOM 2018-IEEE Conference on Computer Communications. Piscataway: IEEE, 2018: 2546-2554.

[50] EYAL I, GENCER A E, SIRER E G, et al. Bitcoin-ng: A scalable blockchain protocol[C]// 13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16). Berkeley: USENIX, 2016: 45-59.

[51] GERVAIS A, KARAME G O, WÜST K, et al. On the security and performance of proof of work blockchains[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2016: 3-16.

[52] BAHRI L, GIRDZIJAUSKAS S. When trust saves enegry-a reference franework for proof-of-trust (pot) blockchains[C]//The Web Conference 2018. New York: ACM, 2018: 1165-1169.

[53] MONTGOMERY C M K, NELSON L, SEN S, et al. Balancing cpu and network in the cell distributed b-tree store[C]//2016 USENIX Annual Technical Conference. Berkeley: USENIX, 2016: 451.

[54] LI F, DAS S, SYAMALA M, et al. Accelerating relational databases by leveraging remote memory and rdma[C]//Proceedings of the 2016 International Conference on Management of Data. New York: ACM, 2016: 355-370.

[55] KALIA A, KAMINSKY M, ANDERSEN D G. Using rdma efficiently for key-value services[C]//Proceedings of the 2014 ACM Conference on SIGCOMM. New York: ACM, 2014: 295-306.

[56] KAMINSKY A K M, ANDERSEN D G. Design guidelines for high performance rdma systems[C]//2016 USENIX Annual Technical Conference. Berkeley: USENIX, 2016: 437.

[57] LIU F, YIN L, BLANAS S. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems[C]//Proceedings of the Twelfth European Conference on Computer Systems. New York: ACM, 2017: 48-63.

[58] LU Y, SHU J, CHEN Y, et al. Octopus: an rdma-enabled distributed persistent memory file system[C]//2017 USENIX Annual Technical Conference (USENIX ATC 17). Berkeley: USENIX, 2017: 773-785.

[59] COSTA P, BALLANI H, RAZAVI K, et al. R2c2: A network stack for rack-scale computers[J]. ACM SIGCOMM Computer Communication Review, 2015, 45(4): 551-564.

[60] NOVAKOVIC S, DAGLIS A, BUGNION E, et al. The case for rackout: Scalable data serving using rack-scale systems[C]//Proceedings of the Seventh ACM Symposium on Cloud Computing. New York: ACM, 2016: 182-195.

[61] NOVAKOVIC S, DAGLIS A, USTIUGOV D, et al. Mitigating load imbalance in distributed data serving with rack-scale memory pooling[J]. ACM Transactions on Computer Systems (TOCS), 2019, 36(2): 1-37.

[62] LUO L, NELSON J, CEZE L, et al. Parameter hub: a rack-scale parameter server for distributed deep neural network training[C]//Proceedings of the ACM Symposium on Cloud Computing. New York: ACM, 2018: 41-54.

[63] NANAVATI M, WIRES J, WARFIELD A. Decibel: Isolation and sharing in disaggregated rack-scale storage[C]//14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17). Berkeley: USENIX, 2017: 17-33.

[64] DAGLIS A, NOVAKOVIĆ S, BUGNION E, et al. Manycore network interfaces for in-memory rack-scale computing[J]. ACM SIGARCH Computer Architecture News, 2015, 43 (3S): 567-579.

[65] LEGTCHENKO S, CHEN N, CLETHEROE D, et al. Xfabric: A reconfigurable in-rack network for rack-scale computers[C]//13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16). Berkeley: USENIX, 2016: 15-29.

[66] TSAI S Y, SHAN Y, ZHANG Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores[C]//2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20). Berkeley: USENIX, 2020: 33-48.

[67] TARIQ A, PAHL A, NIMMAGADDA S, et al. Sequoia: Enabling quality-of-service in serverless computing[C]//Proceedings of the 11th ACM Symposium on Cloud Computing. New York: ACM, 2020: 311-327.

[68] VISWANATHAN R, BALASUBRAMANIAN A, AKELLA A. Network-accelerated distributed machine learning for multi-tenant settings[C]//Proceedings of the 11th ACM Symposium on Cloud Computing. New York: ACM, 2020: 447-461.

[69] SINGHVI A, AKELLA A, GIBSON D, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters[C]//Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication. New York: ACM, 2020: 708-721.

[70] HARCHOL Y, MUSHTAQ A, FANG V, et al. Making edge-computing resilient[C]// Proceedings of the 11th ACM Symposium on Cloud Computing. New York: ACM, 2020: 253-266.

[71] KAUFMANN A, STAMLER T, PETER S, et al. Tas: Tcp acceleration as an os service[C]// Proceedings of the Fourteenth EuroSys Conference 2019. New York: ACM, 2019: 1-16.

[72] COMMUNITY D. Data plane development kit[EB/OL]. 2020. https://www.dpdk.org/.

[73] BELAY A, PREKAS G, KLIMOVIC A, et al. {IX}: A protected dataplane operating system for high throughput and low latency[C]//11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). Berkeley: USENIX, 2014: 49-65.

[74] FOUNDATION L. What is the vector packet processor (vpp)[EB/OL]. 2020. https://fd.io/docs/vpp/master/.

[75] MARTY M, DE KRUIJF M, ADRIAENS J, et al. Snap: a microkernel approach to host networking[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York: ACM, 2019: 399-413.

[76] OUSTERHOUT A, FRIED J, BEHRENS J, et al. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads[C]//16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19). Berkeley: USENIX, 2019: 361-378.

[77] YU S, LIU M, DOU W, et al. Networking for big data: A survey[J]. IEEE Communications Surveys & Tutorials, 2017, 19(1): 531-549.

[78] TSAI S Y, ZHANG Y. Lite kernel rdma support for datacenter applications[C]//Proceedings of the 26th Symposium on Operating Systems Principles. New York: ACM, 2017: 306-324.

[79] MELLANOX TECHNOLOGIES I. Infiniband in the enterprise data center[EB/OL]. 2006. http://www.mellanox.com/pdf/whitepapers/InfiniBand__EDS.pdf.

[80] ISLAM N S, WASI-UR RAHMAN M, LU X, et al. High performance design for hdfs with byte-addressability of nvm and rdma[C]//Proceedings of the 2016 International Conference on Supercomputing. New York: ACM, 2016: 8.

[81] WASI-UR RAHMAN M, LU X, ISLAM N S, et al. High-performance design of yarn mapreduce on modern hpc clusters with lustre and rdma[C]//Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International. Piscataway: IEEE, 2015: 291-300.

[82] CHEN J, LI K, TANG Z, et al. A parallel random forest algorithm for big data in a spark cloud computing environment[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(4): 919-933.

[83] PENG S, WANG G, ZHOU Y, et al. An immunization framework for social networks through big data based influence modeling[J]. IEEE Transactions on Dependable and Secure Computing, 2017.

[84] MELLANOX TECHNOLOGIES I. Mellanox ofed for linux user manual[EB/OL]. 2018. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v4_3.pdf.

[85] LU X, RAHMAN M W U, ISLAM N, et al. Accelerating spark with rdma for big data processing: Early experiences[C]//2014 IEEE 22nd Annual Symposium on High-Performance Interconnects. Piscataway: IEEE, 2014: 9-16.

[86] MITCHELL C, MONTGOMERY K, NELSON L, et al. Balancing {CPU} and network in the cell distributed b-tree store[C]//2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16). Berkeley: USENIX, 2016: 451-464.

[87] KALIA A, KAMINSKY M, ANDERSEN D G. Fasst: Fast, scalable and simple distributed transactions with two-sided ({RDMA}) datagram rpcs[C]//12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). Berkeley: USENIX, 2016: 185-201.

[88] WEI X, SHI J, CHEN Y, et al. Fast in-memory transaction processing using rdma and htm[C]//Proceedings of the 25th Symposium on Operating Systems Principles. Berkeley: USENIX, 2015: 87-104.

[89] WEI X, DONG Z, CHEN R, et al. Deconstructing rdma-enabled distributed transactions: Hybrid is better![C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). Berkeley: USENIX, 2018: 233-251.

[90] POKE M, HOEFLER T. Dare: High-performance state machine replication on rdma networks[C]//Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. New York: ACM, 2015: 107-118.

[91] WANG C, JIANG J, CHEN X, et al. Apus: Fast and scalable paxos on rdma[C]//Proceedings of the 2017 Symposium on Cloud Computing. New York: ACM, 2017: 94-107.

[92] CHEN Y, WEI X, SHI J, et al. Fast and general distributed transactions using rdma and htm[C]//Proceedings of the Eleventh European Conference on Computer Systems. New York: ACM, 2016: 26.

[93] AKIYAMA S, TAURA K. Uni-address threads: scalable thread management for rdma-based work stealing[C]//Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. New York: ACM, 2015: 15-26.

[94] MONGODB I. Nosql databases explained[EB/OL]. 2018. https://www.mongodb.com/nosql-explained.

[95] MANGO D. Top 5 considerations when evaluating nosql databases[J]. White Paper.

[96] HAN J, HAIHONG E, LE G, et al. Survey on nosql database[C]//2011 6th international conference on pervasive computing and applications. Piscataway: IEEE, 2011: 363-366.

[97] ABADI D, MADDEN S, FERREIRA M. Integrating compression and execution in column-oriented database systems[C]//Proceedings of the 2006 ACM SIGMOD international conference on Management of data. New York: ACM, 2006: 671-682.

[98] GEORGE L. Hbase: the definitive guide: random access to your planet-size data[M]. [S.l.]: " O'Reilly Media, Inc.", 2011.

[99] WEBBER J. A programmatic introduction to neo4j[C]//Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity. New York: ACM, 2012: 217-218.

[100] SU M, ZHANG M, CHEN K, et al. Rfp: When rpc is faster than server-bypass with rdma[C]// Proceedings of the Twelfth European Conference on Computer Systems. New York: ACM, 2017: 1-15.

[101] HUANG J, OUYANG X, JOSE J, et al. High-performance design of hbase with rdma over infiniband[C]//2012 IEEE 26th International Parallel and Distributed Processing Symposium. Piscataway: IEEE, 2012: 774-785.

[102] Blockchain on aws[EB/OL]. Amazon Web Services, Inc, 2018. https://aws.amazon.com/blockchain/.

[103] Azure blockchain service[EB/OL]. Microsoft Azure, 2018. https://azure.microsoft.com/en-us/services/blockchain-service/.

[104] Ibm blockchain platform[EB/OL]. IBM Cloud Technologies, 2018. https://www.ibm.com/cloud/blockchain-platform.

[105] Bitcoin-wikipedia[EB/OL]. 2018. https://en.wikipedia.org/wiki/Bitcoin.

[106] Ethereum[EB/OL]. Ethereum, Inc, 2015. https://www.ethereum.org/.

[107] ANDROULAKI E, BARGER A, BORTNIKOV V, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains[C]//Proceedings of the Thirteenth EuroSys Conference. New York: ACM, 2018: 30.

[108] DANIELLARIMER. eosio | blockchain software architecture[EB/OL]. 2019. https://eos.io/.

[109] XU Z, HAN S, CHEN L. Cub, a consensus unit-based storage scheme for blockchain system[C]//2018 IEEE 34th International Conference on Data Engineering (ICDE). Piscataway: IEEE, 2018: 173-184.

[110] SUKHWANI H, MARTÍNEZ J M, CHANG X, et al. Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric)[C]//Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on. Piscataway: IEEE, 2017: 253-255.

[111] LARIMER D. Delegated proof-of-stake (dpos)[J]. Bitshare whitepaper, 2014.

[112] Bitcoin.com | bitcoin news and technology source[EB/OL]. https://www.bitcoin.com/.

[113] ZHENG Z, XIE S, DAI H, et al. An overview of blockchain technology: Architecture[J]. Consensus, and Future Trends, 2017, 10.

[114] HUANG B, JIN L, LU Z, et al. Rdma-driven mongodb: An approach of rdma enhanced nosql paradigm for large-scale data processing[J]. Information Sciences, 2019, 502: 376-393.

[115] ALI M, NELSON J C, SHEA R, et al. Blockstack: A global naming and storage system secured by blockchains.[C]//USENIX Annual Technical Conference. Berkeley: USENIX, 2016: 181-194.

[116] BARBORAK M, DAHBURA A, MALEK M. The consensus problem in fault-tolerant computing[J]. ACM Computing Surveys (CSur), 1993, 25(2): 171-220.

[117] TAYLOR L O, MCKEE M, LAURY S K, et al. Induced-value tests of the referendum voting mechanism[J]. Economics Letters, 2001, 71(1): 61-65.

[118] GATTERMAYER J, TVRDIK P. Blockchain-based multi-level scoring system for p2p clusters[C]//Parallel Processing Workshops (ICPPW), 2017 46th International Conference on. Piscataway: IEEE, 2017: 301-308.

[119] AITZHAN N Z, SVETINOVIC D. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams[J]. IEEE Transactions on Dependable and Secure Computing, 2018, 15(5): 840-852.

[120] LIN I C, LIAO T C. A survey of blockchain security issues and challenges.[J]. IJ Network Security, 2017, 19(5): 653-659.

[121] CHRISTIDIS K, DEVETSIKIOTIS M. Blockchains and smart contracts for the internet of things[J]. Ieee Access, 2016, 4: 2292-2303.

[122] Hyperledger –open source blockchain technologies[EB/OL]. The Linux Foundation, 2018. https://www.hyperledger.org/.

[123] DUA A, BULUSU N, FENG W C, et al. Towards trustworthy participatory sensing[C]// Proceedings of the 4th USENIX conference on Hot topics in security. Berkeley: USENIX, 2009: 8-8.

[124] HU S, CAI C, WANG Q, et al. Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization[C]//IEEE INFOCOM 2018-IEEE Conference on Computer Communications. Piscataway: IEEE, 2018: 792-800.

[125] CHEN H, CHEN R, WEI X, et al. Fast in-memory transaction processing using rdma and htm[J]. ACM Transactions on Computer Systems (TOCS), 2017, 35(1): 3.

[126] TARIQ M A, KOLDEHOFE B, ROTHERMEL K. Securing broker-less publish/subscribe systems using identity-based encryption[J]. IEEE transactions on parallel and distributed systems, 2013, 25(2): 518-528.

[127] ANUSREE P, SREEDHAR S. A security framework for brokerless publish subscribe system using identity based signcryption[C]//2015 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015]. Piscataway: IEEE, 2015: 1-5.

[128] SHITOLE S, GUJAR A. Securing broker-less publisher/subscriber systems using cryptographic technique[C]//2016 International Conference on Computing Communication Control and automation (ICCUBEA). Piscataway: IEEE, 2016: 1-6.

[129] DAUBERT J, FISCHER M, GRUBE T, et al. Anonpubsub: Anonymous publish-subscribe overlays[J]. Computer Communications, 2016, 76: 42-53.

[130] ION M, RUSSELLO G, CRISPO B. Design and implementation of a confidentiality and access control solution for publish/subscribe systems[J]. Computer networks, 2012, 56(7): 2014-2037.

[131] YANG K, ZHANG K, JIA X, et al. Privacy-preserving attribute-keyword based data publish-subscribe service on cloud platforms[J]. Information Sciences, 2017, 387: 116-131.

[132] ZHAO Y, LI Y, MU Q, et al. Secure pub-sub: Blockchain-based fair payment with reputation for reliable cyber physical systems[J]. IEEE Access, 2018, 6: 12295-12303.

[133] KREPS J, NARKHEDE N, RAO J, et al. Kafka: A distributed messaging system for log processing[C]//Proceedings of the NetDB: volume 11. [S.l.: s.n.], 2011: 1-7.

[134] MATRIX B. Eos node tool[EB/OL]. https://eosnode.tools/blocks.

[135] Tpu pods[EB/OL]. https://cloud.google.com/tpu/.

[136] WEI X, XIE X, CHEN R, et al. Characterizing and optimizing remote persistent memory with rdma and nvm[C]//2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21). Berkeley: USENIX, 2021: 523-536.

[137] ZHANG X, ZHENG X, WANG Z, et al. High-density multi-tenant bare-metal cloud[C]// Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 483-495.

[138] Hp moonshot system[EB/OL]. http://bit.ly/1mZD4yJ.

[139] Seamicro sm15000 fabric compute systems[EB/OL]. http://bit.ly/1hQepIh.

[140] NEUGEBAUER R, ANTICHI G, ZAZO J F, et al. Understanding pcie performance for end host networking[C]//Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. New York: ACM, 2018: 327-341.

[141] REGULA J. Using non-transparent bridging in pci express systems[J]. PLX Technology, Inc, 2004, 31.

[142] COMMUNITY L K. Ntb drivers in linux kernel[EB/OL]. 2020. https://www.kernel.org/doc/Documentation/ntb.txt.

[143] LI B, CUI T, WANG Z, et al. Socksdirect: Datacenter sockets can be fast and compatible[M]// Proceedings of the ACM Special Interest Group on Data Communication. New York: ACM, 2019: 90-103.

[144] MELLANOX. Messaging accelerator (vma)[EB/OL]. 2019. https://github.com/mellanox/libvma.

[145] COMMUNITY R. Redis data structure store[EB/OL]. 2020. https://redis.io/.

[146] COMMUNITY R. Rocksdb[EB/OL]. 2020. https://rocksdb.org/.

[147] TU S, ZHENG W, KOHLER E, et al. Speedy transactions in multicore in-memory databases[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. New York: ACM, 2013: 18-32.

[148] SHAN Y, TSAI S Y, ZHANG Y. Distributed shared persistent memory[C]//Proceedings of the 2017 Symposium on Cloud Computing. Piscataway: IEEE, 2017: 323-337.

[149] YANG J, IZRAELEVITZ J, SWANSON S. Filemr: Rethinking {RDMA} networking for scalable persistent memory[C]//17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20). Berkeley: USENIX, 2020: 111-125.

[150] XUE J, MIAO Y, CHEN C, et al. Fast distributed deep learning over rdma[C]//Proceedings of the Fourteenth EuroSys Conference 2019. New York: ACM, 2019: 1-14.

[151] JIANG Y, ZHU Y, LAN C, et al. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters[C]//14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20). Berkeley: USENIX, 2020: 463-479.

[152] BARTHELS C, LOESING S, ALONSO G, et al. Rack-scale in-memory join processing using rdma[C]//Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2015: 1463-1475.

[153] FOUNDATION A S. Apache spark streaming[EB/OL]. 2018. https://spark.apache.org/streaming/.

[154] VENKATARAMANI V, AMSDEN Z, BRONSON N, et al. Tao: how facebook serves the social graph[C]//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2012: 791-792.

[155] KULKARNI C, MOORE S, NAQVI M, et al. Splinter: Bare-metal extensions for multi-tenant low-latency storage[C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). Berkeley: USENIX, 2018: 627-643.

[156] DU D, YU T, XIA Y, et al. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting[C]//Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 467-481.

[157] HOEFLER T, ROSS R B, ROSCOE T. Distributing the data plane for remote storage access[C]//15th Workshop on Hot Topics in Operating Systems (HotOS {XV}). New York: ACM, 2015.

[158] INTEL. Intel rack scale design[EB/OL]. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[159] ASANOVIĆ K. Firebox: A hardware building block for 2020 warehouse-scale computers[J]. 2014.

[160] PETERSEN C. Introducing lightning: A flexiblenvme jbof[EB/OL]. 2016. https://code.facebook.com/posts/989638804458007/introducinglightning-a-flexible-nvme-jbof/.

[161] EMC. Dssd d5[EB/OL]. 2016. https://www.emc.com/enus/storage/flash/dssd/dssd-d5/index.htm.

[162] WANG Q, LU Y, XU E, et al. Concordia: Distributed shared memory with in-network cache coherence[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). Berkeley: USENIX, 2021: 277-292.

[163] CHO S, CARTER A, EHRLICH J, et al. Moolle: Fan-out control for scalable distributed data stores[C]//2016 IEEE 32nd International Conference on Data Engineering (ICDE). Piscataway: IEEE, 2016: 1206-1217.

[164] ZHU Y, GHOBADI M, MISRA V, et al. Ecn or delay: Lessons learnt from analysis of dcqcn and timely[C]//Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies. New York: ACM, 2016: 313-327.

[165] INTEL. Intel® 64 and ia-32 architectures software developer's manual volume 1: Basic architecture[EB/OL]. 2021. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-1-basic-architecture.html.

[166] Pci express base 4.0[EB/OL]. 2014. https://xdevs.com/doc/Standards/PCI/PCI_Express_Base_4.0_Rev0.3_February19-2014.pdf.

[167] INTEL. Intel® 64 and ia-32 architectures optimization reference manual[EB/OL]. 2020. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[168] INTEL. Intel® data direct i/o technology[EB/OL]. 2020. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[169] FARSHIN A, ROOZBEH A, MAGUIRE JR G Q, et al. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks[C]//2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20). Berkeley: USENIX, 2020: 673-689.

[170] JINGJING W, OMKAR M. Dpdk pmd for ntb[EB/OL]. Intel, 2019. https://static.sched.com/hosted_files/dpdkna2019/35/DKPMDforPCleNon-TransparentBridge.pptx.

[171] FITZPATRICK B. Distributed caching with memcached[J]. Linux journal, 2004, 124.

[172] FOUNDATION T A S. ab - apache http server benchmarking tool[EB/OL]. 2020. https://httpd.apache.org/docs/2.4/programs/ab.html.

[173] KIM D, YU T, LIU H H, et al. Freeflow: Software-based virtual {RDMA} networking for containerized clouds[C]//16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19). Berkeley: USENIX, 2019: 113-126.